(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 1 265 161 A2

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
11.12.2002 Bulletin 2002/50

(51) Int Cl.[7]: **G06F 17/30**

(21) Application number: **02253371.5**

(22) Date of filing: **14.05.2002**

(84) Designated Contracting States:
AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE TR
Designated Extension States:
AL LT LV MK RO SI

(30) Priority: **04.06.2001 US 874788**

(71) Applicant: **Hewlett-Packard Company**
Palo Alto, CA 94304 (US)

(72) Inventors:
- **Baskins, Douglas L.**
  Fort Collins, Colorado 80524 (US)
- **Silverstein, Alan**
  Fort Collins, Colorado 80525 (US)

(74) Representative: **Jehan, Robert et al**
Williams Powell
4 St Paul's Churchyard
London EC4M 8AY (GB)

(54) System for and method of storing data

(57) An adaptive digital tree data structure incorporates a rich pointer object (104, 105, 110-112, 114-118), the rich pointer including both conventional address redirection information (116B) used to traverse the structure and supplementary information (116A) used to optimize tree traversal, skip levels, detect errors, and store state information. The structure of the pointer is flexible so that, instead of storing pointer information, data may be stored in the structure of the pointer itself and thereby referenced without requiring further redirection.

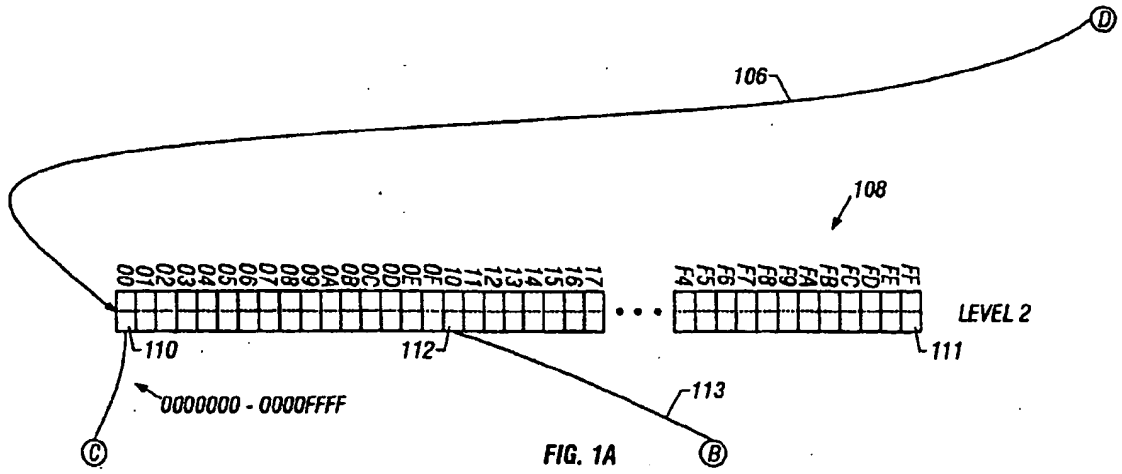


FIG. 1A

EP 1 265 161 A2

Description

[0001] The present invention relates generally to the field of data structures, preferably to a hierarchical data organization in which the structure of the data organization is dependent on the data stored and information is associated with pointers.

[0002] The present invention is related to co-pending, European Patent Application No.(RJ/N12198), (RJ/N12199), and (RJ/N12336), filed the same day as this application.

[0003] Computer processors and associated memory components continue to increase in speed. As hardware approaches physical speed limitations, however, other methods for generating appreciable decreases in data access times are required. Even when such limitations are not a factor, maximizing software efficiency maximizes the efficiency of the hardware platform, extending the capabilities of the hardware/software system as a whole. One method of increasing system efficiency is by providing effective data management, achieved by the appropriate choice of data structure and related storage and retrieval algorithms. For example, various prior art data structures and related storage and retrieval algorithms have been developed for data management including arrays, hashing, binary trees, AVL trees (height-balanced binary trees), b-trees, and skiplists. In each of these prior art data structures, and related storage and retrieval algorithms, an inherent trade-off has existed between providing faster access times and providing lower memory overhead. For example, an array allows for fast indexing through the calculation of the address of a single array element but requires the pre-allocation of the entire array in memory before a single value is stored, and unused intervals of the array waste memory resources. Alternatively, binary trees, AVL trees, b-trees and skiplists do not require the pre-allocation of memory for the data structure and attempt to minimize allocation of unused memory but exhibit an access time which increases as the population increases.

[0004] An array is a prior art data structure that has a simplified structure and allows for rapid access of the stored data. However, memory must be allocated for the entire array and the structure is inflexible. An array value is looked up "positionally," or "digitally," by multiplying the index by the size (e.g., number of bytes) allocated to each element of the array and adding the offset of the base address of the array. Typically, a single Central Processing Unit (CPU) cache line fill is required to access the array element and value stored therein. As described and typically implemented, the array is memory inefficient and relatively inflexible. Access, however, is provided as O(1), *i.e.*, independent of the size of the array (ignoring disk swapping).

[0005] Alternatively, other data structures previously mentioned including binary trees, b-trees, skiplists and hash tables, are available which are more memory efficient but include undesirable features. For example, hashing is used to convert sparse, possibly multi-word indexes (such as strings) into array indexes. The typical hash table is a fixed-size array, and each index into it is the result of a hashing algorithm performed on the original index. However, in order for hashing to be efficient, the hash algorithm must be matched to the indexes which are to be stored. Hash tables also require every data node to contain a copy of (or a pointer to) the original index (key) so you can distinguish nodes in each synonym chain (or other type of list). Like an array, use of hashing requires some preallocation of memory, but it is normally a fraction of the memory that must be allocated for a flat array, if well designed *i.e.*, the characteristics of the data to be stored are well known, behaved and matched to the hashing algorithm, collision resolution technique and storage structure implemented.

[0006] In particular, digital trees, or tries, provide rapid access to data, but are generally memory inefficient. Memory efficiency may be enhanced for handling sparse index sets by keeping tree branches narrow, resulting in a deeper tree and an increase in the average number of memory references, indirections, and cache line fills, all resulting in slower access to data. This latter factor, *i.e.*, maximizing cache efficiency, is often ignored when such structures are discussed yet may be a dominant factor affecting system performance. A trie is a tree of smaller arrays, or branches, where each branch decodes one or more bits of the index. Prior art digital trees have branch nodes that are arrays of simple pointers or addresses. Typically, the size of the pointers or addresses are minimized to improve the memory efficiency of the digital tree.

[0007] At the "bottom" of the digital tree, the last branch decodes the last bits of the index, and the element points to some storage specific to the index. The "leaves" of the tree are these memory chunks for specific indexes, which have application-specific structures.

[0008] Digital trees have many advantages including not requiring memory to be allocated to branches which have no indexes or zero population (also called an empty subexpanse). In this case the pointer which points to the empty subexpanse is given a unique value and is called a null pointer indicating that it does not represent a valid address value. Additionally, the indexes which are stored in a digital tree are accessible in sorted order which allows identification of neighbors. An "expanse" of a digital tree as used herein is the range of values which could be stored within the digital tree, while the population of the digital tree is the set of values that are actually stored within the tree. Similarly, the expanse of a branch of a digital tree is the range of indexes which could be stored within the branch, and the population of a branch is the number of values (*e.g.*, count) which are actually stored within the branch. (As used herein, the term "population" refers to either the set of indexes or the count of those indexes, the meaning of the term being apparent to those skilled in the art from the context in which the term is used.)

[0009] "Adaptive Algorithms for Cache-Efficient Trie Search" by Acharya, Zhu and Shen (1999), the disclosure of which is hereby incorporated herein by reference, describes cache-efficient algorithms for trie search. Each of the algorithms use different data structures, including a partitioned-array, B-tree, hashtable, and vectors, to represent different nodes in a trie. The data structure selected depends on cache characteristics as well as the fanout of the node. The algorithms further adapt to changes in the fanout at a node by dynamically switching the data structure used to represent the node. Finally, the size and the layout of individual data structures is determined based on the size of the symbols in the alphabet as well as characteristics of the cache(s). The publication further includes an evaluation of the performance of the algorithms on real and simulated memory hierarchies.

[0010] Other publications known and available to those skilled in the art describing data structures include *Fundamentals of Data Structures in Pascal*, 4th Edition; Horowitz and Sahni; pp 582-594; *The Art of Computer Programming*, Volume 3; Knuth; pp 490-492; Algorithms in C; Sedgewick; pp 245-256, 265-271; "Fast Algorithms for Sorting and Searching Strings"; Bentley, Sedgewick; "Ternary Search Trees"; 5871926, INSPEC Abstract Number: C9805-6120-003; Dr Dobb's Journal; "Algorithms for Trie Compaction", ACM Transactions on Database Systems, 9(2):243-63, 1984; "Routing on longest-matching prefixes"; 5217324, INSPEC Abstract Number: B9605-6150M-005, C9605-5640-006; "Some results on tries with adaptive branching"; 6845525, INSPEC Abstract Number: C2001-03-6120-024; "Fixed-bucket binary storage trees"; 01998027, INSPEC Abstract Number: C83009879; "DISCS and other related data structures"; 03730613, INSPEC Abstract Number: C90064501; and "Dynamical sources in information theory: a general analysis of trie structures"; 6841374, INSPEC Abstract Number: B2001-03-6110-014, C2001-03-6120-023, the disclosures of which are hereby incorporated herein by reference.

[0011] An enhanced storage structure is described in U.S. Patent Application Serial No. 09/457,164 filed December 8, 1999, entitled "A FAST EFFICIENT ADAPTIVE, HYBRID TREE," (the '164 application) assigned in common with the instant application and hereby incorporated herein by reference in its entirety. The data structure and storage methods described therein provide a self-adapting structure which self-tunes and configures "expanse" based storage nodes to minimize storage requirements and provide efficient, scalable data storage, search and retrieval capabilities. The structure described therein, however, does not take full advantage of certain sparse data situations.

[0012] An enhancement to the storage structure described in the '164 application is detailed in U.S. Patent Application Serial No. 09/725,373, filed November 29, 2000, entitled "A DATA STRUCTURE AND STORAGE AND RETRIEVAL METHOD SUPPORTING ORDINALITY BASED SEARCHING AND DATA RETRIEVAL", assigned in common with the instant application and hereby incorporated herein by reference in its entirety. This latter application describes a data structure and related data storage and retrieval method which rapidly provides a count of elements stored or referenced by a hierarchical structure of ordered elements (e.g., a tree), access to elements based on their ordinal value in the structure, and identification of the ordinality of elements. In an ordered tree implementation of the structure, a count of indexes present in each subtree is stored, i.e., the cardinality of each subtree is stored either at or associated with a higher level node pointing to that subtree or at or associated with the head node of the subtree. In addition to data structure specific requirements (e.g., creation of a new node, reassignment of pointers, balancing, etc.) data insertion and deletion includes steps of updating affected counts. Again, however, the structure fails to take full advantage of certain sparse data situations.

[0013] The present invention seeks to provide improved data processing, in the preferred embodiments to optimize performance characteristics of a digital tree and similar structures.

[0014] According to an aspect of the present invention there is provided a data structure as specified in claim 1.

[0015] According to another aspect of the present invention there is provided a method of storing indexes in a data structure as specified in claim 6.

[0016] According to another aspect of the present invention there is provided a computer memory as specified in claim 10.

[0017] The preferred system includes a data structure which is stored in the memory, can be treated as a dynamic array, and is accessed through a root pointer. For an empty tree, this root pointer is null, otherwise it points to the first of a hierarchy of branch nodes. Each branch node consists of a plurality of informational or "rich" pointers which subdivide the expanse of the index (key) used to access the data structure. Each rich pointer contains auxiliary information in addition to, or in some cases instead of, the address of (that is, the pointer to) a subsidiary (child) branch or leaf node. This auxiliary information permits various optimizations that result in a positive "return on investment" despite the space required to store the information.

[0018] An informational pointer may contain an address (the actual pointer to a child branch or leaf node); index digits (parts of keys) that help skip levels in the tree or bring leaf information to the present level; population counts that help rapidly count the numbers of valid (stored) indexes in the tree or in any subexpanse (range of indexes); and type information about the next level in the tree, if any, to which the pointer points. Pointers may also provide information for verifying operation and data integrity, and correcting errors. State information may also be bundled with pointers so that the resultant rich pointers provide state information. In this case, the data

structure not only provides a means to store and manipulate data, but includes facilities supporting the processes using the structure. The inclusion of this information allows the digital tree to be compressed in various ways that make it smaller, more cache-efficient, and faster to access and modify, even as the branch nodes are potentially no longer simply arrays of pointers to subsidiary nodes. This information also provides structure and redundancies that allow for faster access to and modification of the tree, as well as detection of data corruption.

[0019] Embodiments of the present invention are described below, by way of example only, with reference to the accompanying drawings, in which:

FIGURES 1A - 1E depict a digital tree which includes a comparison between prior art pointers and an informational pointer for skipping levels in a data structure;

FIGURE 2 is generalized diagram of an informational pointer incorporating immediate storage of indexes;

FIGURE 3 is a chart showing typical storage capabilities of informational pointers used to store immediate indexes;

FIGURES 4A-4D are diagrams of rich pointers used to store 3, 2 and 1 byte immediate indexes on a 32-bit system;

FIGURES 5A-5H are diagrams of rich pointers used to store 7-1 byte immediate indexes on a 64-bit system;

FIGURES 6A - 6D are diagrams of rich pointers used to store immediate indexes and associated values on a 64-bit system;

FIGURES 7A - 7E depict a digital tree which includes a comparison between indexes stored in leaf nodes and informational pointers used as immediate indexes; and

FIGURE 8 is a block diagram of a computer system in which the data structure may be implemented.

[0020] As previously described, typical digital trees exhibit several disadvantages. These disadvantages include memory allocated to null pointers associated with empty branches while exhibiting an increased number of memory references or indirections, and possibly cache line fills, as the size (*i.e.*, "fanout") of the branches narrows to reduce the number of these null pointers. These disadvantages associated with digital trees have limited their use in prior computer applications.

[0021] The described embodiment combines the advantages of the digital tree with smarter approaches to handling both non-terminal nodes (branches) and terminal nodes (leaves) in the tree. These smarter approaches minimize both memory space and processing time, for both lookups, insertions and modifications of data stored in the data structure. Additionally, it ensures the data structure remains efficient as indexes are added or deleted from the data structure. The approaches used by this embodiment include forms of data compression and compaction and help reduce the memory required for the data structure, minimize the number of cache line fills required, and reduce access and retrieval times.

[0022] The described system replaces the simple pointers typically implemented in digital trees with "rich" pointers (herein termed "informational pointers" and used interchangeably therewith) which associate additional information with the redirection or address information of the pointers. This additional information may be used by the data structure and/or by processes accessing the structure. The use of rich pointers within the digital tree permits various optimizations within the data structure. In a preferred embodiment of the invention each rich pointer in a digital tree branch includes multiple segments or portions, typically occupying two words (dependent upon the target platform). The rich pointer may contain an address (the actual pointer), index digits (parts of keys), population counts, type information concerning the next level to which the pointer "points" or is directed to within the tree, redundant data supporting error detection, state information, etc.

[0023] One type of a rich pointer is a narrow-expanse pointer. In particular, one type of data compression that may be used when an expanse is populated by a "dense cluster" of indexes that all have some leading bits in common is supported by a narrow-expanse pointer. to the present invention. The typical representation of the common bits through multiple digital tree branches (or redundant bits in leaves) can be replaced by uniquely representing (e.g., encoding) the common bits as part of or associated with the pointer to the branch or leaf. In a preferred embodiment of the present invention, this type of data compression is limited to common leading whole bytes. The common bits are stored in a rich pointer and the pointer type indicates the level of and the number of remaining undecoded digits in the next object. The remaining undecoded digits imply the number of levels skipped by the narrow pointer. The rich pointer is stored (*i.e.*, associated) with the pointer to the next level, which has an expanse smaller than it would otherwise. Preferably, each subexpanse pointer contains a "decode" field that holds all index bytes decoded so far except for the first byte. Narrow pointers provide a method to skip levels within a digital tree which save memory used to store the indexes and reduces the number of memory references and cache fills required.

[0024] FIGURES 1A - 1E depict the use of narrow-expanse pointers in a digital tree. (For the purposes of the present illustration, examples of the data structure are given with reference to a 32-bit word size platform wherein indexes are single words (as opposed, *e.g.*, to character strings of arbitrary length) although it is understood that the system is not so limited and, to the contrary, encompasses other word sizes and configurations including, but not limited to 16, 32, 64 and 128-bit word sizes.) As used herein, the term "slot" refers to a

record or group of cells of an array associated with, and/or including a pointer to a child node or, more generally, a subexpanse of indexes, together with any data associated with the pointer. Generally, the array is "indexed" so that each cell or "slot" is associated with an offset value corresponding to an ordinal value of the slot within the array. Thus, in further detail, root pointer node 101 is used for accessing the underlying data structure of the digital tree. Root pointer node 101 includes address information 102 diagrammatically shown as an arrow pointing to a first or "top" level node 103, in this illustration, a branch node. (Note, the terminology used herein labels the top node of a tree pointed to by the root as "level 1", children of the level 1 node are designated as "level 2" nodes, etc. According to this convention, the level of any branch or leaf node is equal to one more than the number of digits (bytes) decoded in the indexes stored above that node. It is further noted that this convention, while representative, is for purposes of the present explanation and other conventions may be adopted including, for example, designating leaf nodes as constituting a first level of the tree. In this latter case, a preferred embodiment of the invention, the level of any branch or leaf node is equal to the number of digits (bytes) remaining to decode in the indexes stored at or below that node.) First level node 103 includes slots or enhanced pointer arrays for up to 256 lower level nodes and represents the entire expanse of the data structure, *i.e.* indexes 00000000 through FFFFFFFF hex by implementing a 256-way branch. (Note that, although a preferred embodiment decodes 1 byte of the index at each branch, other divisions of the index may be used including, for example, decoding 4 bits to implement a 16-way branch at each level of the tree, etc.) First level node 103 includes first slot 104 (containing an adaptable object) which corresponds to expanse 00000000-00FFFFFF and last slot 105 which corresponds to a final expanse portion including indexes FF000000-FFFFFFFF. The pointer contained in the pointer field in slot 104 points to a first one of 256 of the next level subexpanses (level 2 in the digital tree) while the pointer in slot 105 points to the most significant upper 1/256th of level 2.

[0025] The first subexpanses of level 2 includes subsidiary node 108 in turn including an array of 256 pointers directed to lower level nodes 119 and 120. As shown, the expanse covered by node 108 (*i.e.*, an index range of 00000000-00FFFFFF hex) is only sparsely populated by indexes falling within the subexpanse ranges covered by third level nodes 119 and 120 (*i.e.*, 00000000-0000FFFF and 00100000-0010FFFF hex, respectively). Thus, while the pointers in slots 110 and 112 include valid redirection information to (*i.e.*, address of) nodes 119 and 120, the remaining 254 pointers of node 108, including the pointer in slot 111 covering an uppermost expanse range of 00FF0000-00FFFFFF hex, are null pointers, *i.e.*, have a special value reserved for pointers that are not directed to any target location or to empty nodes. Note that node 120 is similarly sparsely populated, with all indexes falling within a single subexpanse node 121 associated with a range of 00100200-001002FF hex and pointed to by the sole active pointer in node 120, that is pointer 122. Thus, not only does node 120 require the allocation of additional storage space for 256 pointers, but access to indexes referenced by it to leaf nodes requires two indirections and therefore two cache fills.

[0026] Thus, as pictured, slot 110 contains a pointer to a level 3 slot which corresponds to 000000-0000FFFF. Additionally, slot 112 contains a pointer which points to a separate subexpanse 120 of level 3 which correlates to 00100000-0010FFFF. Similarly, slots within level 3 may further point to a subexpanse at level 4. Operationally, level 4 of FIGURES 1A - 1E is reached by consecutive decoding of one-byte portions of the index and traversing the tree in accordance with the decoded values. The first one byte (00) is used to identify slot 104 which contains the corresponding pointer to traverse the tree from level 1 to the corresponding portion of level 2 *i.e.*, the node addressed by the pointer of slot 104. The next byte (10) is used to identify slot 112 which contains the corresponding pointer to traverse the tree from node 108 to subsidiary node 120 at level 3. The next byte (02) is used to identify slot 122 which contains the corresponding pointer to traverse the tree from node 120 of level 3 to node 121 of level 4. Once at level 4, the remaining byte is used to access the appropriate slot of node 121 to retrieve the data associated with the index value. As described, this process requires four separate memory references and potentially four different cache fills to identify the correct memory address which corresponds to the index.

[0027] If an expanse, or subexpanse, is sparsely populated with a small number of dense clusters of subsidiary indexes, a rich pointer may be used to encode the common bits of the populated subexpanse or indexes. Still referring to FIGURES 1A - 1E, the upper 1/256 subexpanses of level 2 subsidiary node 109 contains a dense cluster of indexes which each lie within the range of FF100200-FF1002FF. The other portions of the upper 1/256 subexpanse, FF100000-FF1001FF and FF100300-FF10FFFF do not contain indexes. In this case, a rich pointer can be used to point directly to the level 4 portion of the subexpanse, skipping level 3 and eliminating the need for a memory reference or indirection to level 3. Specifically, the corresponding slot 116 contains a rich pointer node which includes an information data field 116A and a pointer node 116B to the next subexpanse or other structure for accessing the subsidiary indexes. The information data field 116A includes the common bytes (*i.e.*, index portion) of the remaining indexes, 02, because the remaining indexes all fall within the range of FF100200-FF1002FF.

[0028] In this case, the rich pointer is used to eliminate one of the memory references and possibly one cache fill. The first two bytes (FF) of the index are used to

traverse from the level 1 of the tree to the appropriate portion of level 2. Once at the level 2 node the rich pointer is used to traverse from the level 2 node directly to the level 4 node.

[0029] The rich pointer structure encompasses at least two types of rich pointers or adaptable objects including a pointer type as described above and an immediate type. The immediate type supports immediate leaves or immediate indexes. That is, when the population of an expanse is relatively sparse, a rich pointer is used to store the indexes "immediately" within a digital tree branch, rather requiring traversal of the digital tree down to the lowest level to access the index. This format is akin to the "immediate" machine instruction format wherein an instruction specifies an immediate operand which immediately follows any displacement bytes. Thus, an immediate index or a small number of indexes are stored in the node, avoiding one or more redirections otherwise required to traverse the tree and arrive at some distant leaf node. Immediate indexes thereby provide a way of packing small populations (or small number of indexes) directly into a rich pointer structure instead of allocating more memory and requiring multiple memory references and possible cache fills to access the data.

[0030] A two word format of the preferred embodiment readily supports the inclusion of immediate indexes. Within the rich pointer, this is accomplished by storing index digits in the information data field. A rich pointer implemented in a 32-bit system may store anywhere from a single 3-byte immediate index up to seven 1-byte indexes, while a rich pointer in a 64-bit system may store up to 15 1-byte immediate indexes. The generalized structure of a rich pointer (also referred to as an adaptable object) supporting immediate indexes is shown in FIGURE 2. The rich pointer includes one or more indexes "I", depending on the word-size of the platform and the size of the index, and an 8-bit Type field that also encodes the index size and the number of immediate indexes.

[0031] As mentioned, the number of immediate indexes stored will depend upon the word-size of the indexes, upper levels within the tree nearest the root requiring larger indexes, smaller indexes being found as the tree is traversed toward the leaves. Examples of numbers of immediate index values of various sizes accommodated by 32-bit and 64-bit machines according to a preferred embodiment are presented in FIGURE 3 wherein indexes are mapped to valid/invalid indicators and have no associated values. FIGURES 4A-4D illustrate 3, 2 and 1-byte index sizes stored in an immediate rich pointer structure implemented on a 32-bit platform, while FIGURES 5A-5H illustrate index sizes of 7 through 1 byte implemented on a 64-bit machine. The structures of FIGURES 4A-4D and 5A-5H are also directed to an embodiment of the invention in which only the presence or absence of an index is indicated without any other value being associated with the indexes.

[0032] FIGURES 6A - 6D illustrate another embodiment of the invention on a 64-bit machine wherein a value is associated with each index I_n. According to this embodiment, when a single immediate index I_1 of up to 7 bytes is stored in a rich pointer structure, a 64-bit value associated with the index is also stored as shown in FIGURE 6A. However, if more than one immediate index is to be stored, such as when an index may be represented by 3-bytes, 2-bytes or 1-byte indexes (FIGURES 6B - 6C, respectively), then the first 8-byte word of the rich pointer is instead used as a pointer to values associated with the respective multiple indexes. A similar configuration is used to store values associated with indexes when the invention is implemented on a 32-bit machine.

[0033] Immediate indexes are packed into rich pointers starting at the "first byte" (farthest from the type field), and possibly leaving some unused storage. An exception is present in a preferred embodiment wherein, if a single immediate index is stored, the indexes begin at the first byte of the second word to allow the first word to be a value area corresponding to the index, for those arrays that map an index to a value (see FIGURES 4A, 5A and 6A). The structure of an ordinary leaf and the indexes portion of a rich pointer containing an immediate index are identical once the starting address, index size, and population are known.

[0034] Thus, as described, an immediate index rich pointer structure may be thought of as including a small leaf. Such a structure is particularly helpful to represent a sparsely populated expanse where the indexes reside in the rich pointer itself.

[0035] FIGURES 7A - 7E illustrate a comparison between typical pointers and a rich pointer which can be used to store immediate indexes. The indexes would typically be stored in the portion of level 4 node 121 in the corresponding array cell or "slot." By using a rich pointer as an immediate index, the indexes that would otherwise reside in a leaf node such as leaf node 701 are instead stored in the corresponding portion of a higher level node, e.g., level 2 node 109. For a 64 bit system, one or more indexes can be stored in slot 116 of level 2 node 109 in immediate index data field 702. As diagramed, slot 116 is logically divided into multiple subslots, each storing an immediate index. The use of rich pointers as immediate indexes avoids at least one memory reference and one or more cache line fills.

[0036] Another use of informational fields available with rich pointers is directed to storing state information associated with the object referenced by the pointer or otherwise describing and/or storing state information such as the state of the procedure accessing the structure. Thus, while the tree itself is not a state machine, when combined with a specified index to insert, delete, or retrieve, it may be used as input to the accessing process that allows the code to operate similar to a state machine. Each tree subexpanse pointer includes a "type" field (*e.g.*, 8-bits) that encodes one of a large number (*e.g.*, 256) of enumerated object types. Levels

in the tree are directly encoded in the enumerations. Rich pointers allow rich pointer types which are a large number of very specific next-level object types. The tree traversal code can be thought of as a state machine whose inputs are the index/key to decode and the nodes of the tree. In a preferred embodiment, the state machine algorithm provides software which appears to be a single large switch, allowing the software to act as a collection of small, fast code chunks, each pre-optimized to perform one task well with minimum run-time computations. Since the tree level is encoded in each pointer type, the traversal code does not need to track the current tree level. Instead, this state information is stored within the nodes of the tree itself.

[0037] Rich pointers may also be used to detect errors in data, interpret and process data, etc., by providing information redundancy. That is, data characterizing the information stored as part of a rich pointer may be used to detect errors in the referenced data much as an error detection code might be used. This information may also be used to confirm, for example, position in a tree by encoding level or similar information in association with each pointer to form a rich pointer.

[0038] In particular, in practice it is not feasible or desirable to compress all unused bits out of a rich pointer. Machine instruction efficiency is partially dependent on word and byte boundaries. While the ratio of cache fill time to CPU instruction time is sufficiently high that cache efficiency is important, it is generally still low compared to other data compression methods that, for example, are directed to minimizing disk reads. (Cache-efficient programs must balance CPU time against "complete" data compression.) The result of "incomplete" compression is to provide and use some redundant data in rich pointers that allows tree traversal code to opportunistically, but very "cheaply," detect and report many types of data corruption in the tree itself, resulting either from tree management code defects or external accidents. In the preferred embodiment, cheaply detected corruptions may result in a void pointer return, while "expensive" detections result in assertion failure in debug code only and are ignored in production code. The unused bits of a rich pointer may be used to opportunistically determine various types of data corruption. As used herein, error detection data refers to any redundant data available to detect data corruption, whether or not that data is stored solely for the purpose of error detection or stored for other (functional) purposes but having a secondary use to detect errors.

[0039] For example, an error condition may be identified by checking to see that a pointer type matches the tree level. For example, it is inappropriate for certain objects such as a "Leaf!" object to appear at other than the lowest level of the tree furthest from the root. With reference to FIGURES 7A - 7E, if the type field 703 contains an invalid value, such as 255, an invalid rich pointer type would be indicated and appropriate error processing performed.

[0040] Another check is performed for decode bytes in subexpanse pointers which include already-decoded index bytes that are not required as part of a narrow pointer, but nonetheless must match the path taken to this point in the tree. It is more efficient and simpler to store already-decoded index bytes this way then to somehow optimize to storing only required narrow-pointer bytes.

[0041] Rich pointers also allow computational efficiencies. In particular, when a single immediate index is stored in a rich pointer, there is room (e.g., in the Decode field) to store all but the first byte of the index, not just the remaining undecoded bytes. This allows faster traversal and modification. Like decode bytes, these redundant bytes must agree with the path traversed to the immediate index.

[0042] Rich pointers also support pointer portability. That is, when a narrow-expanse pointer indicates only the level of the subsidiary node, rather than the number of levels being skipped, it remains "portable". Like any rich pointer that "knows" about the object to which it refers but not about the object in which it resides, a portable narrow-expanse pointer allows easier branch insertion and deletion when an "outlier" index is inserted or deleted. (An outlier is an index that belongs under the full subexpanse of the slot occupied by the narrow-expanse pointer, but not under the present narrow expanse of that pointer.)

[0043] FIGURE 8 is a diagram of a computer system capable of supporting and running a memory storage program implementing and maintaining a data structure as taught herein. Thus, although the structure is adaptable to a wide range of data structures, programming languages, operating systems and hardware platforms and systems, FIGURE 8 illustrates one such computer system 800 comprising a platform suitable to support the structure. Computer system 800 includes Central Processing Unit (CPU) 801 coupled to system bus 802. CPU 801 may be any general purpose CPU, such as an HP PA-8500 or Intel Pentium processor. However, the system is not restricted by the architecture of CPU 801 as long as CPU 801 supports the operations as described herein, e.g., the use of pointers. System bus 802 is coupled to Random Access Memory (RAM) 803, which may be SRAM, DRAM or SDRAM. ROM 804 is also coupled to system bus 802, which may be PROM, EPROM, or EEPROM. RAM 803 and ROM 804 hold user and system data and programs as is well known in the art.

[0044] System bus 802 is also coupled to input/output (I/O) controller card 805, communications adapter card 811, user interface card 808, and display card 809. The I/O card 805 connects to storage devices 806, such as one or more of a hard drive, a CD drive, a floppy disk drive, a tape drive, to the computer system. Communications card 811 is adapted to couple computer system 800 to network 812, which may be one or more of a telephone network, a Local (LAN) and/or a Wide-Area

(WAN) network, an Ethernet network, and/or the Internet network and can be wire line or wireless. User interface card 808 couples user input devices, such as keyboard 813 and pointing device 807, to computer system 800. Display card 809 is driven by CPU 801 to control display device 810.

[0045] The disclosures in United States patent application No.09/874,788, from which this application claims priority, and in the abstract accompanying this application are incorporated herein by reference.

Claims

1. A data structure for storage of indexes in a computer memory, including:

   a hierarchy of branch nodes (103, 108, 109, 119, 120, 121, 123) ordered into a plurality of levels beginning with a top level branch (103), each of said branch nodes including an array of adaptable objects (104, 105, 110, 112, 111, 114-118) each associated with a subexpanse of said indexes mapped by a respective one of said branch nodes, said adaptable objects each including a type field (T) indicating a type of said adaptable object, said type including a pointer type in which said adaptable object is configured to include a pointer (116B) to another node and an information data field (116A) configured to store information about said other node, and an immediate type in which at least one of said indexes is stored in said adaptable object.

2. A structure according to claim 1, wherein said information data field (116A) represents an index portion common to subsidiary ones of said indexes such that a subsidiary node (123) is more than one level lower in the data structure than its parent (109) and does not encode a common portion of said subsidiary indexes.

3. A structure according to claim 1 or 2, wherein a pointer field (116B) associated with one of said adaptable objects (116) of said pointer type at one level of said data structure is directed to another of said branch nodes (123) at another level of said data structure that is removed from said one level by at least two levels and said information data field of said adaptable object includes a portion of a plurality of said indexes common to all of said indexes in a subexpanse associated with said adaptable object.

4. A structure according to claim 1, 2 or 3, including a plurality of leaf nodes (701) associated with one or more of the indexes, wherein a pointer field associated with one of said adaptable objects of said pointer type at one level of said data structure is directed to one of said leaf nodes at another level of said data structure that is removed from said one level by at least two levels and said information data field of said adaptable object includes a portion of a plurality of said indexes common to all of said indexes in a subexpanse residing in said one leaf node.

5. A structure according to any preceding claim, wherein at least a part of a aid adaptable object includes an immediate index data field (702) configured to represent at least a portion of at least one subsidiary index, such that said subsidiary index is immediately present without further indirection through a pointer to a different location in said computer memory.

6. A method of storing indexes in a data structure, including the steps of:

   defining a data structure including a hierarchy of branch nodes (103, 108, 109, 119, 120, 121, 123) ordered into a plurality of levels beginning with a top level branch (103), each of said branch nodes including an array of adaptable objects (104, 105, 110, 112, 111, 114-118) each associated with a subexpanse of said indexes mapped by a respective one of said branch nodes, said adaptable objects each including a type field (T) indicating a type of said adaptable object, said type including a pointer type (116B) in which said adaptable object is configured to include a pointer to another node and an information data field (116A) configured to store information about said other node, and an immediate type in which at least one of said indexes is stored in said adaptable object; and storing the indexes in the data structure.

7. A method according to claim 6, including the step of defining said data structure to include a plurality of leaf nodes (701) associated with one or more of the indexes, wherein a pointer field (104) associated with one of said adaptable objects of said pointer type at one level of said data structure is directed to one of said leaf nodes (701) at another level of said data structure that is removed from said one level by at least two levels and said information data field of said adaptable object includes a portion of a plurality of said indexes common to all of said indexes in a subexpanse residing in said one leaf node.

8. A method as in claim 6 or 7, including the step of configuring at least a part of a said adaptable object to include an immediate index data field (702) rep-

resenting at least a portion of at least one subsidiary index, such that said subsidiary index is immediately present without further indirection through a pointer to a different location in said computer memory.

9. A method as in claim 6, 7 or 8, including the steps of storing error detection data (703) in said adaptable objects and detecting an error condition within said data structure using said error detection data.

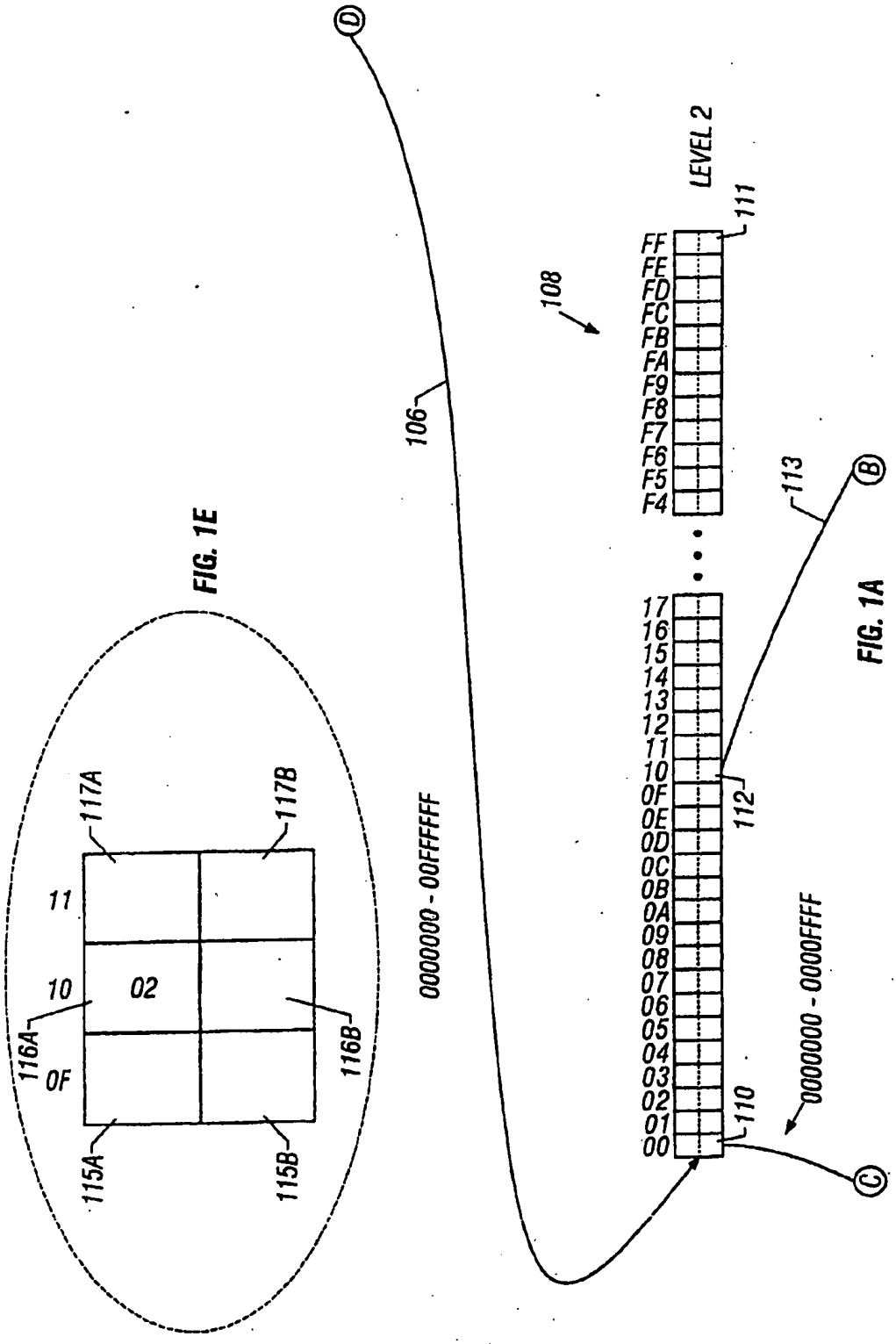
10. A computer memory (803) for storing data for access by an application program executed on a data processing system, including:

a data structure stored in said memory for storage of indexes, said data structure including a hierarchy of branch nodes (103, 108, 109, 119, 120, 121, 123) ordered into a plurality of levels beginning with a top level branch (103), each of said branch nodes including an array of adaptable objects (104, 105, 110, 112, 111, 114-118) each associated with a subexpanse of said indexes mapped by a respective one of said branch nodes, said adaptable objects each including a type field (T) indicating a type of said adaptable object, said type including a pointer type (116B) in which said adaptable object is configured to include a pointer to another node and an information data field (116A) configured to store information about said other node, and an immediate type in which at least one of said indexes is stored in said adaptable object.

EP 1 265 161 A2

FIG. 1A	FIG. 1B
FIG. 1C	FIG. 1D

FIG. 1


D
106
108
LEVEL 2
111
FF
FE
FD
FC
FB
FA
F9
F8
F7
F6
F5
F4
113
B
17
16
15
14
13
12
11
10
0F
0E
0D
0C
0B
0A
09
08
07
06
05
04
03
02
01
00
112
110
0000000 - 00FFFFFF
0000000 - 0000FFFF
C
FIG. 1A
FIG. 1E
117A
117B
11
10
02
0F
116A
116B
115A
115B

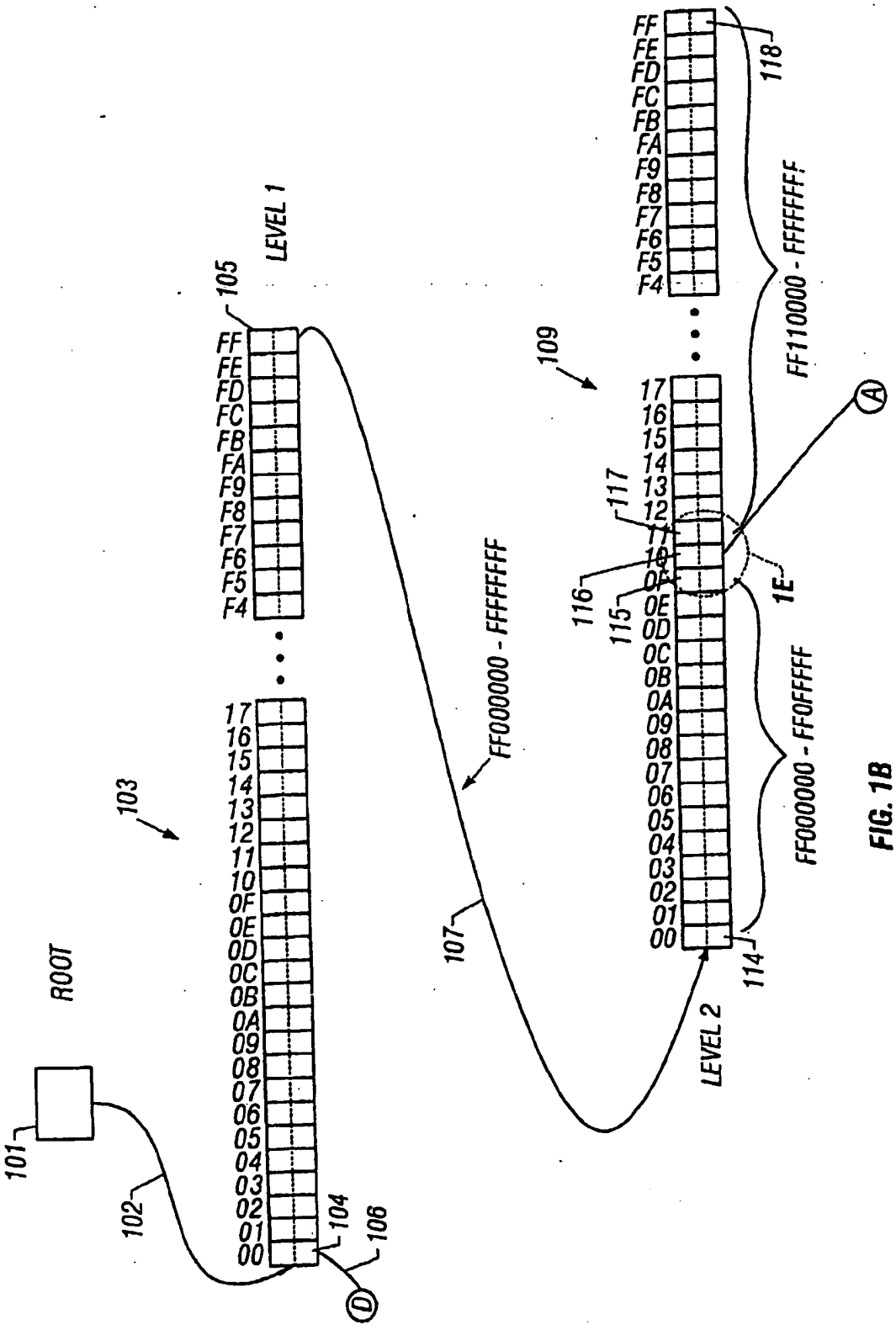

ROOT
101
102
103
LEVEL 1
105
FF
FE
FD
FC
FB
FA
F9
F8
F7
F6
F5
F4
17
16
15
14
13
12
11
10
0F
0E
0D
0C
0B
0A
09
08
07
06
05
04
03
02
01
00
104
106
FF000000 - FFFFFFFF
107
109
LEVEL 2
114
FF110000 - FFFFFFFF
118
FF000000 - FF0FFFFF
116
115
117
1E
A

FIG. 1B

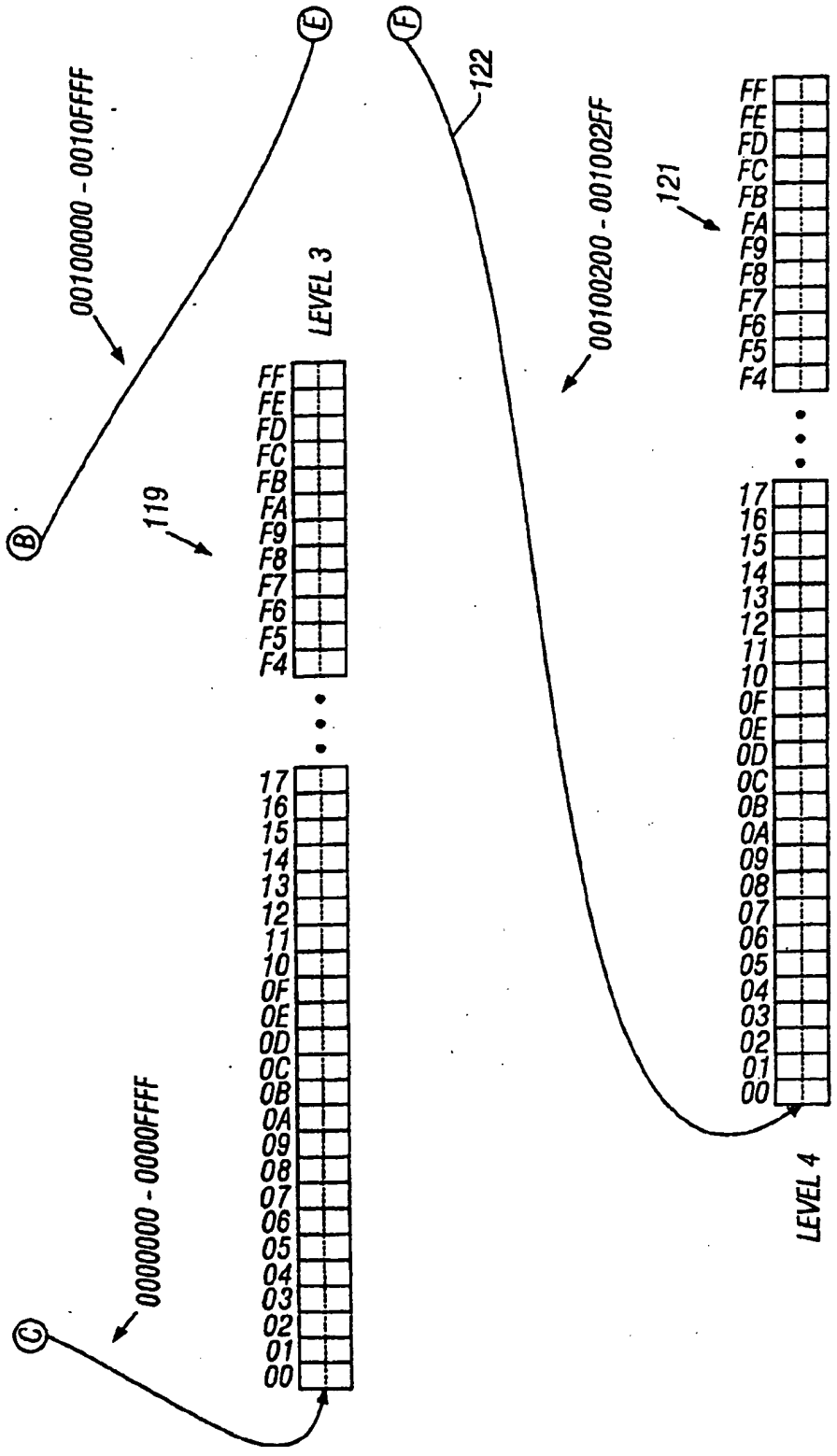

E
F
B
C
00100000 - 0010FFFF
0000000 - 0000FFFF
00100200 - 001002FF
119
121
122
LEVEL 3
LEVEL 4
FF
FE
FD
FC
FB
FA
F9
F8
F7
F6
F5
F4
17
16
15
14
13
12
11
10
0F
0E
0D
0C
0B
0A
09
08
07
06
05
04
03
02
01
00

FIG. 1C

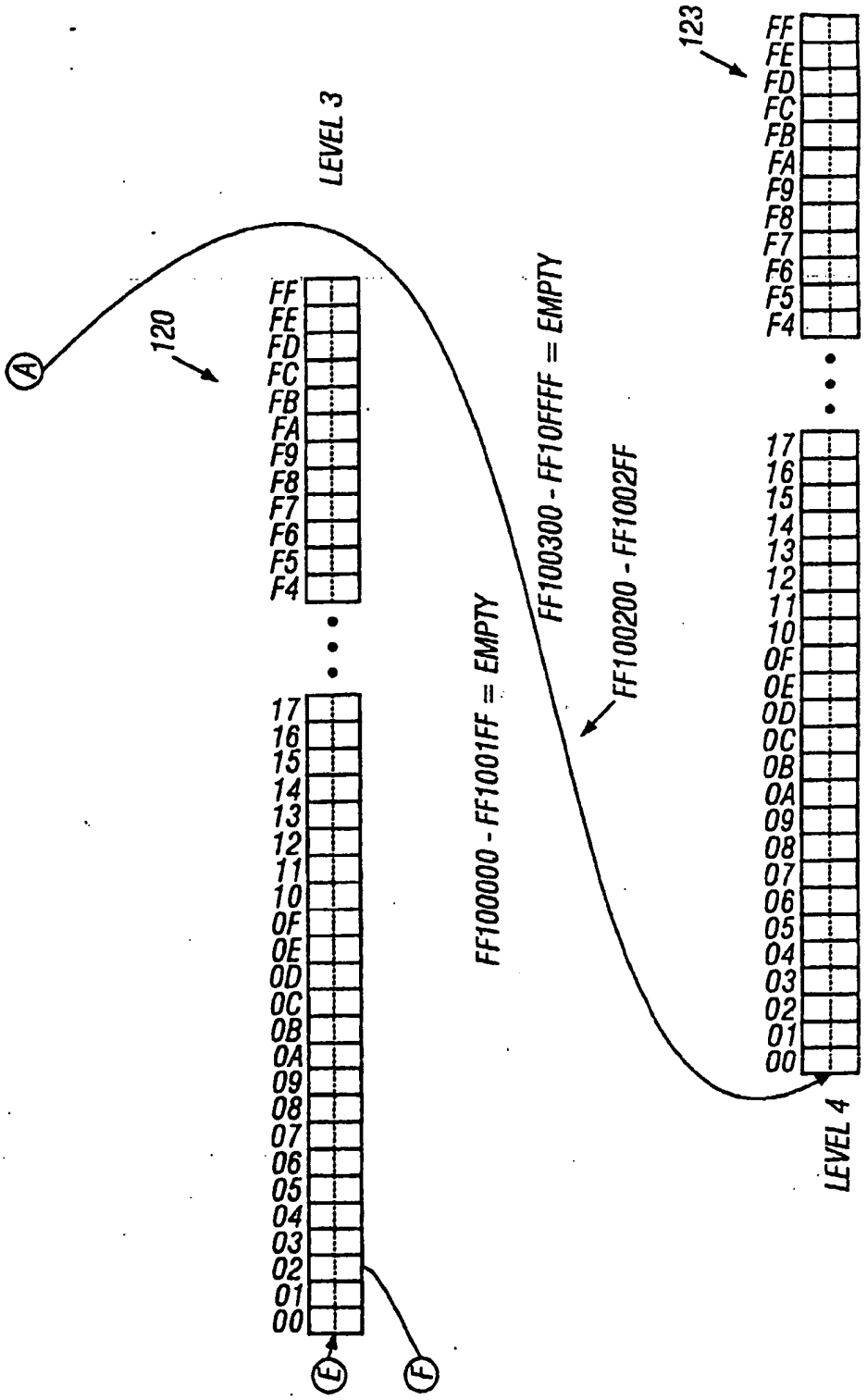
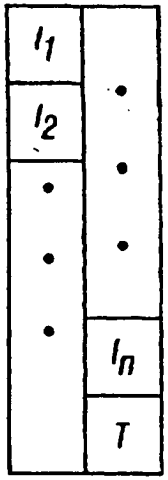

LEVEL 3
120
123
FF100000 - FF1001FF = EMPTY
FF100300 - FF10FFFF = EMPTY
FF100200 - FF1002FF
LEVEL 4
A
E
F
FF FE FD FC FB FA F9 F8 F7 F6 F5 F4
17 16 15 14 13 12 11 10 0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

FIG. 1D



TYPE = 8 BITS

FIG. 2

32-BIT	[64-BIT]	INDEX-SIZE
	[1..2]	[7-BYTE INDEXES]
	[1..2]	[6-BYTE INDEXES]
	[1..3]	[5-BYTE INDEXES]
	[1..3]	[4-BYTE INDEXES]
1..2	[1..5]	3-BYTE INDEXES
1..3	[1..7]	2-BYTE INDEXES
1..7	[1..15]	1-BYTE INDEXES

FIG. 3

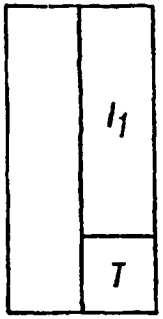


FIG. 4A

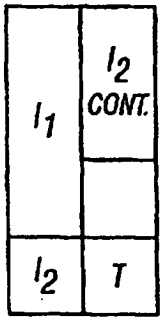


FIG. 4B

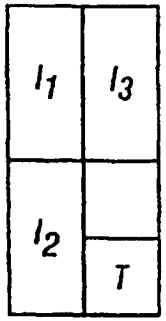


FIG. 4C

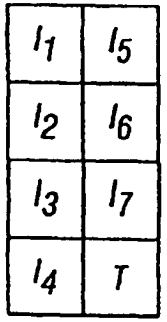


FIG. 4D

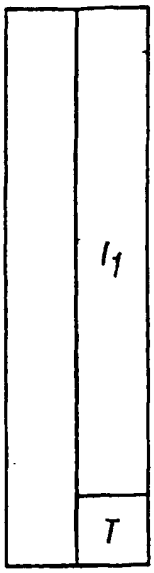


FIG. 5A

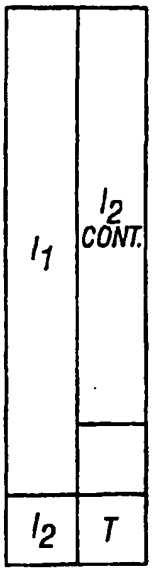


FIG. 5B

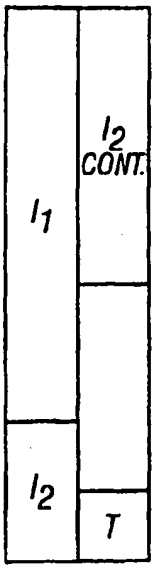


FIG. 5C

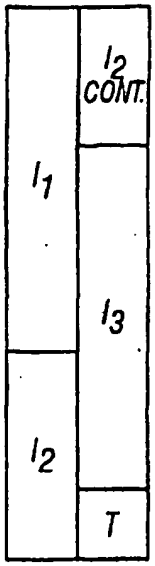


FIG. 5D

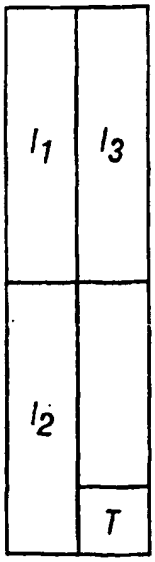


FIG. 5E

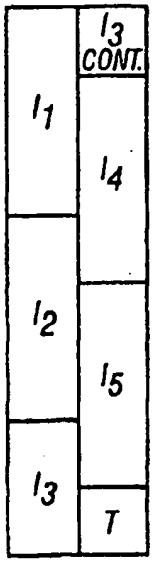


FIG. 5F

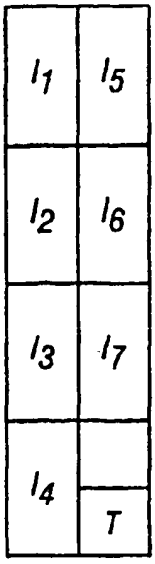


FIG. 5G

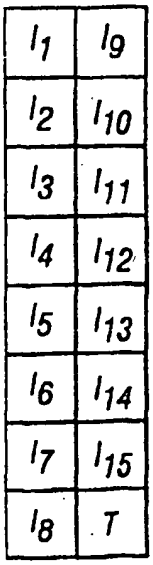


FIG. 5H

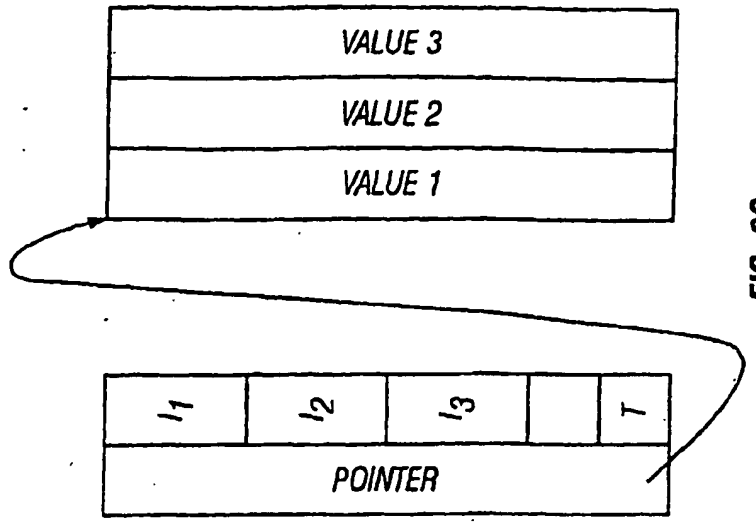


FIG. 6C

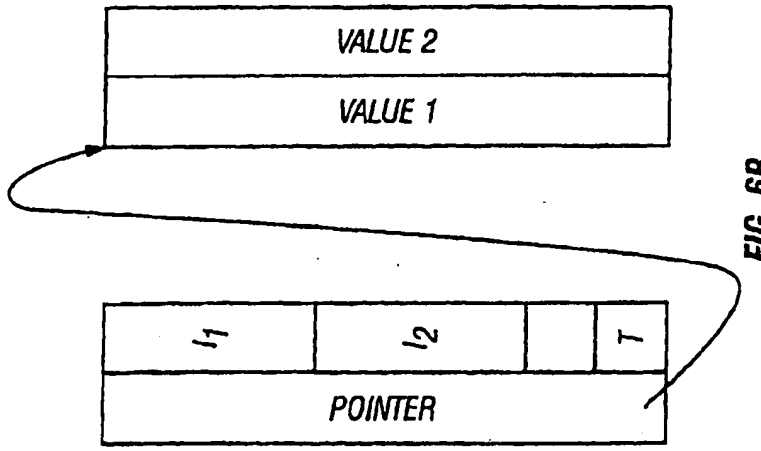


FIG. 6B

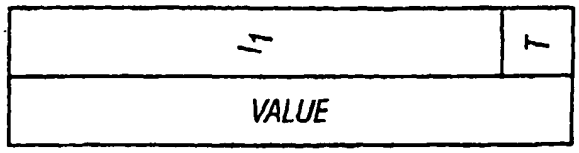


FIG. 6A

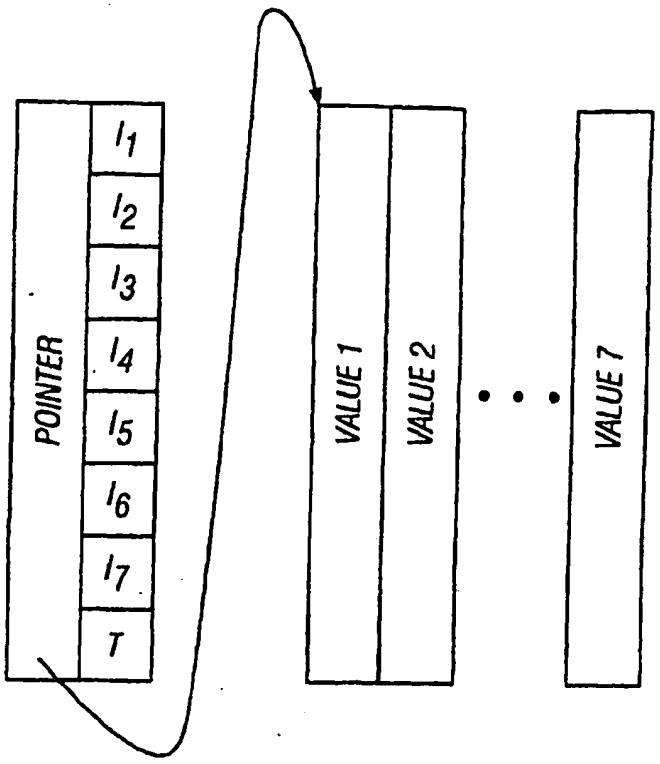


FIG. 6D

FIG.7A	FIG. 7B
FIG. 7C	FIG. 7D

FIG. 7

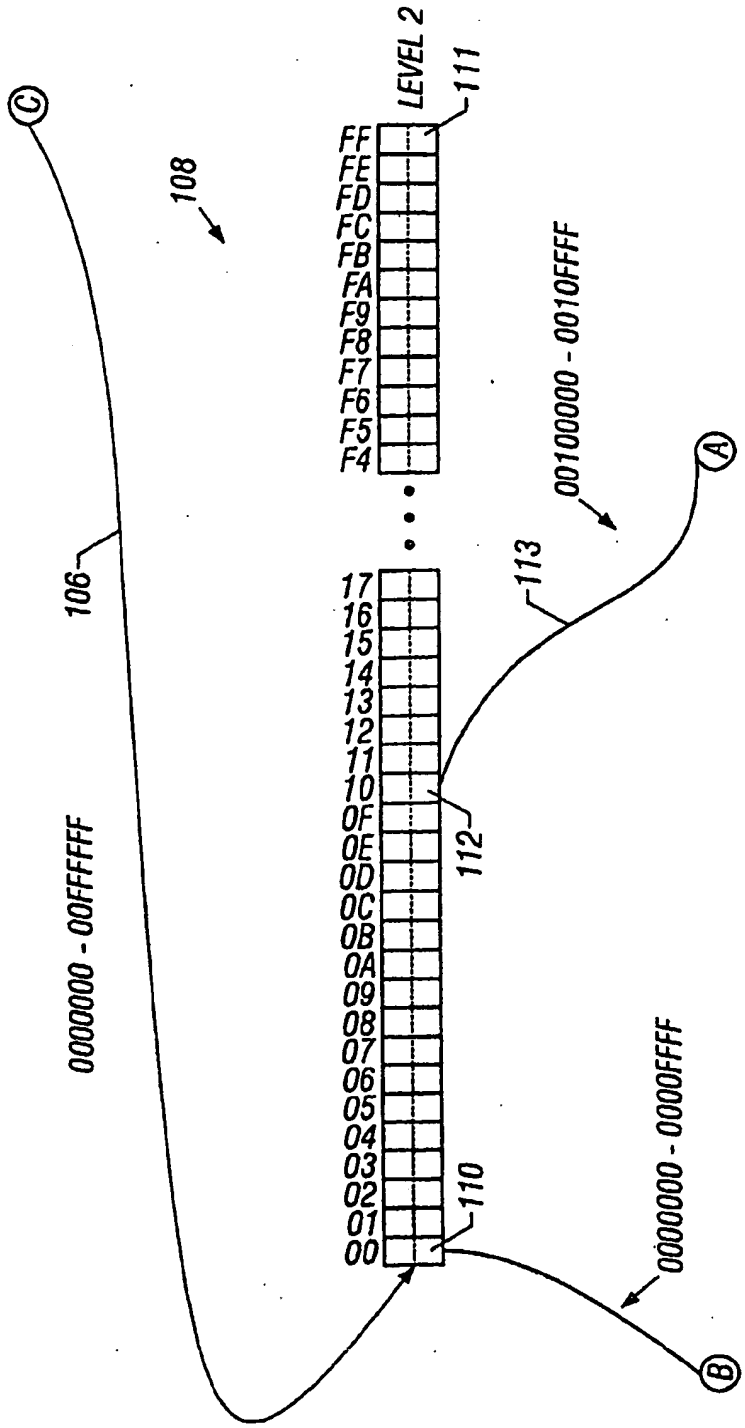

LEVEL 2
111
108
FF
FE
FD
FC
FB
FA
F9
F8
F7
F6
F5
F4
00100000 - 0010FFFF
A
113
106
17
16
15
14
13
12
11
10
0F
0E
0D
0C
0B
0A
09
08
07
06
05
04
03
02
01
00
112
110
0000000 - 00FFFFFF
0000000 - 0000FFFF
C
B

FIG. 7A

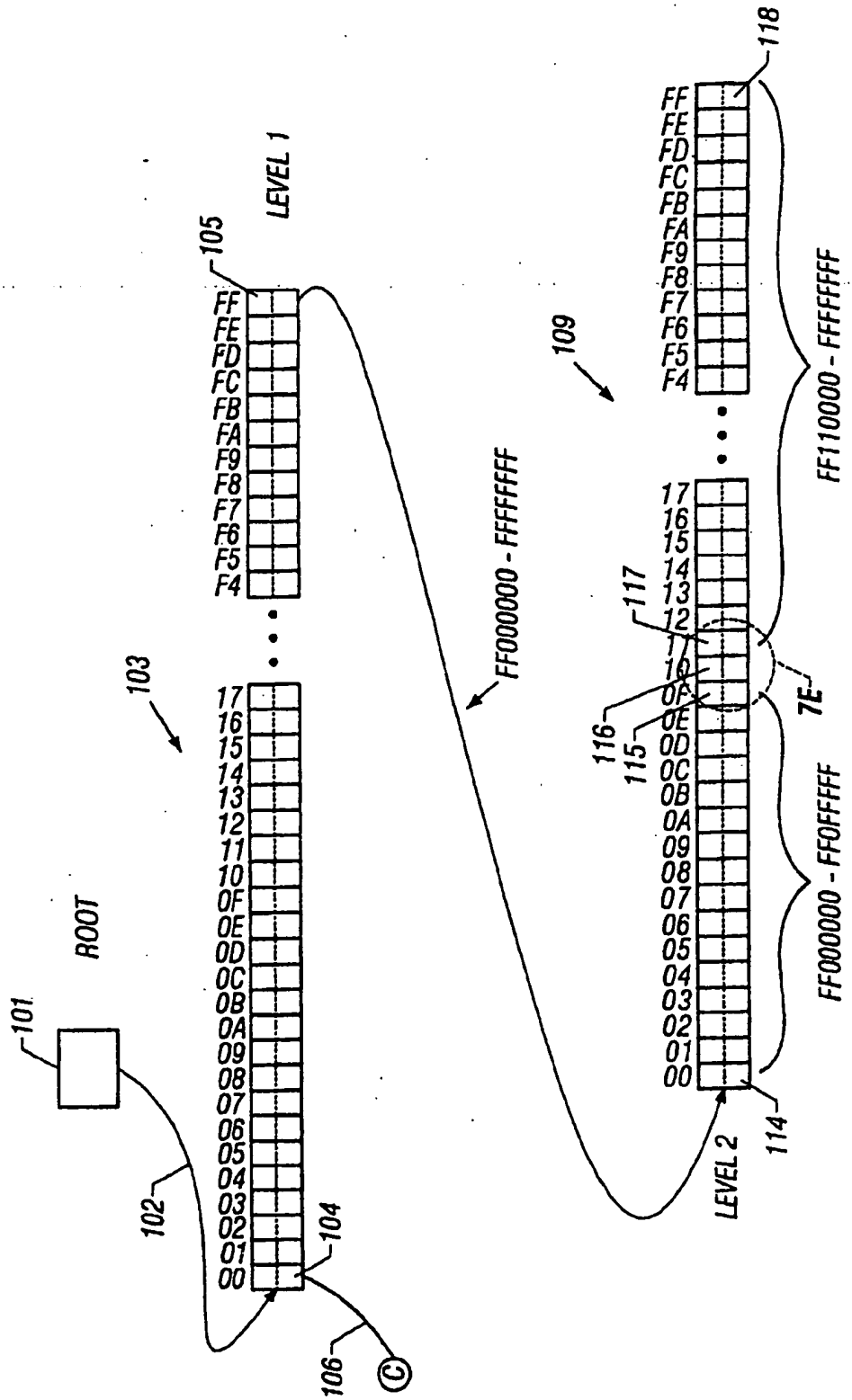

ROOT
101
102
103
104
105
LEVEL 1
106
FF000000 - FFFFFFFF
109
LEVEL 2
114
115
116
117
118
FF000000 - FF0FFFFF
FF110000 - FFFFFFFF
7E

FIG. 7B

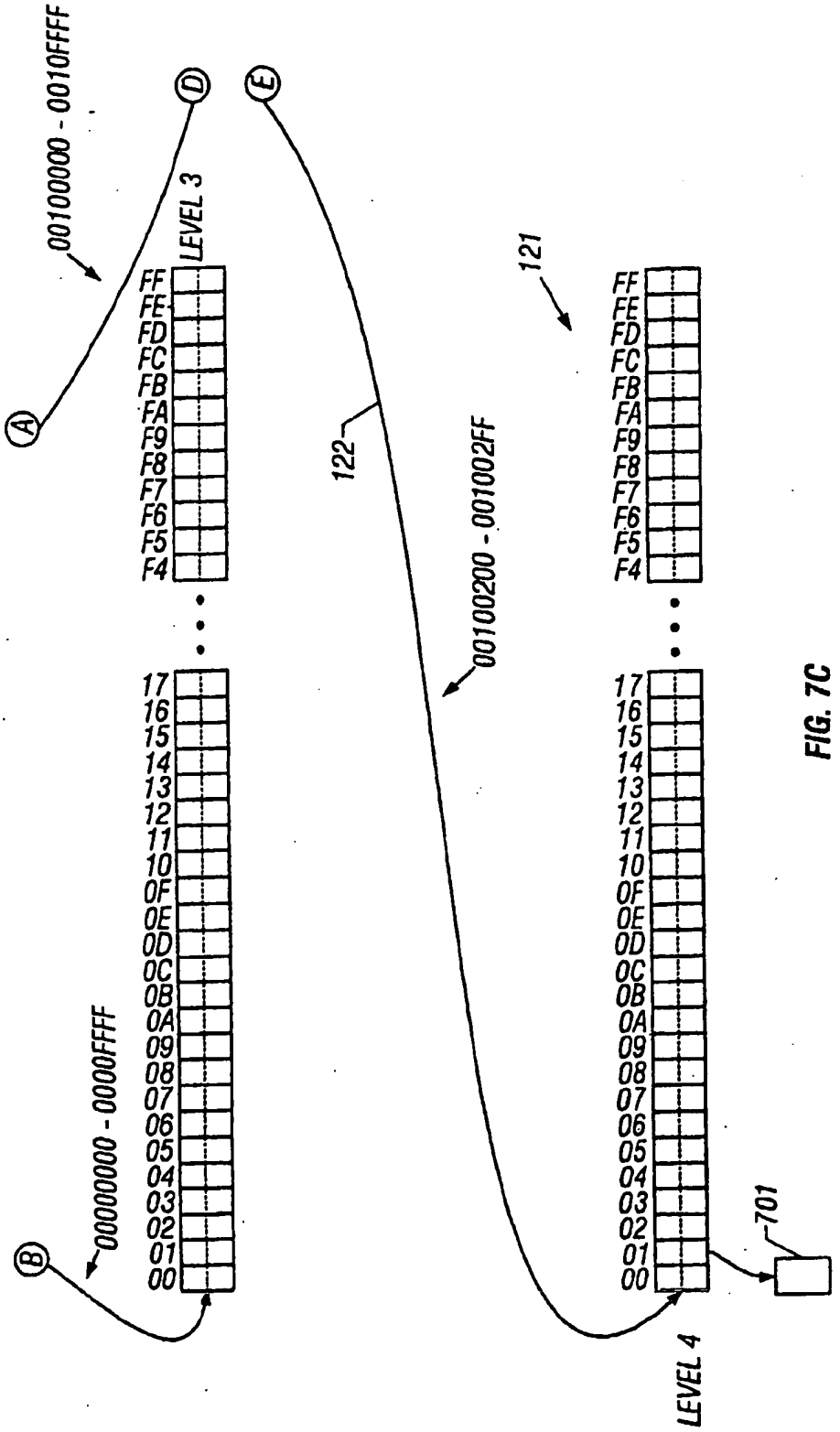

00100000 - 0010FFFF
D
E
A
LEVEL 3
FF
FE
FD
FC
FB
FA
F9
F8
F7
F6
F5
F4
121
122
00100200 - 001002FF
17
16
15
14
13
12
11
10
0F
0E
0D
0C
0B
0A
09
08
07
06
05
04
03
02
01
00
B
00000000 - 0000FFFF
701
LEVEL 4

FIG. 7C

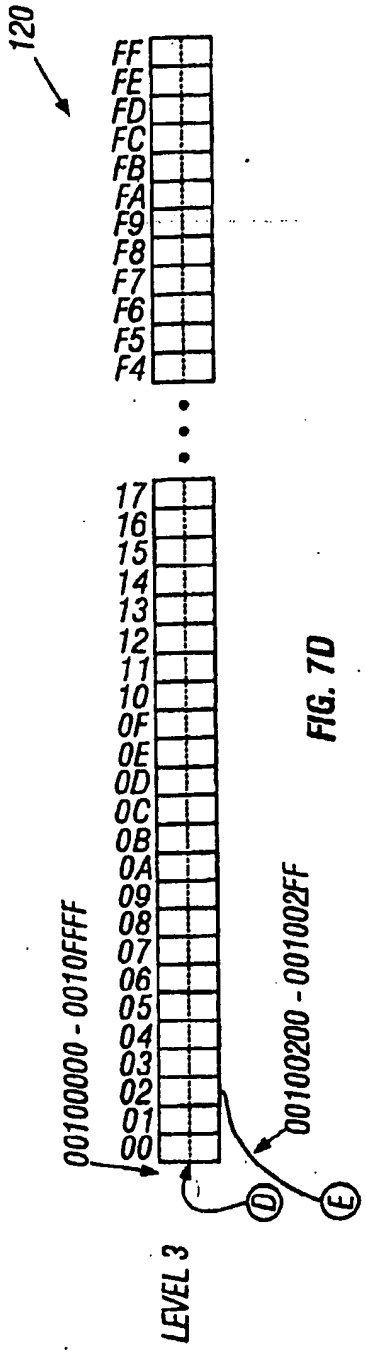

120
FF
FE
FD
FC
FB
FA
F9
F8
F7
F6
F5
F4
17
16
15
14
13
12
11
10
0F
0E
0D
0C
0B
0A
09
08
07
06
05
04
03
02
01
00
00100000 - 0010FFFF
00100200 - 001002FF
LEVEL 3

FIG. 7D

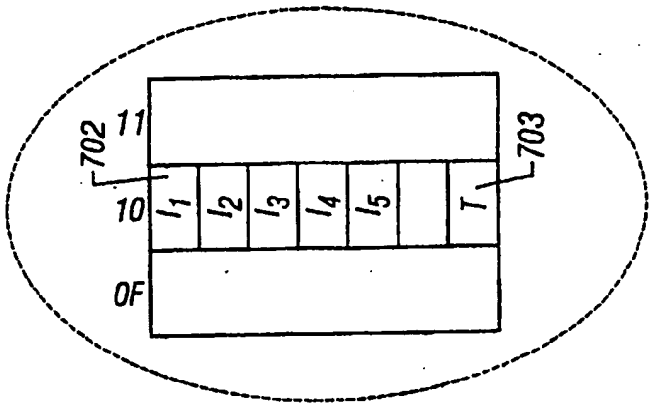

11
10
0F
702
703
I1
I2
I3
I4
I5
T

FIG. 7E

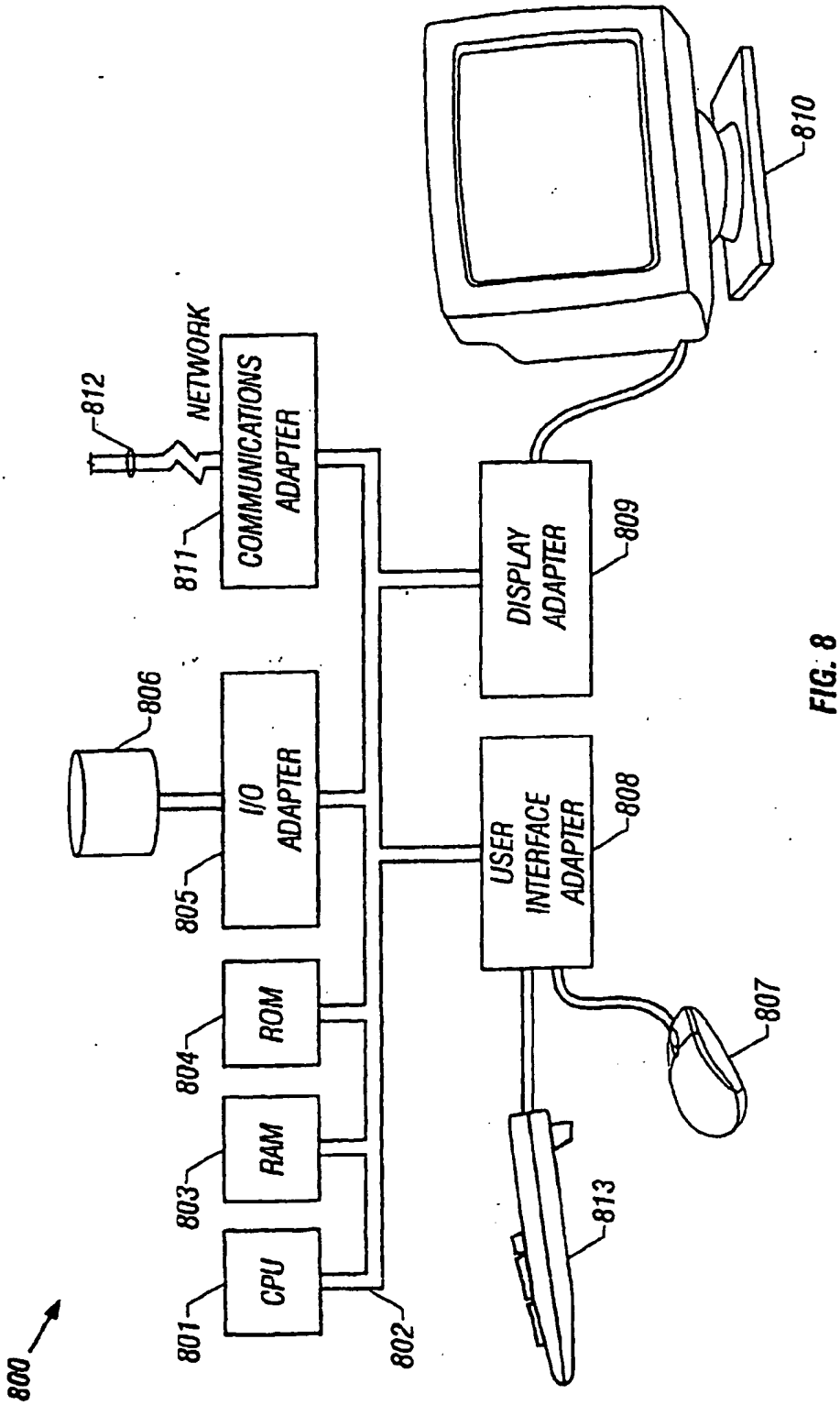

800
801
CPU
803
RAM
804
ROM
805
I/O ADAPTER
806
811
COMMUNICATIONS ADAPTER
812
NETWORK
802
809
DISPLAY ADAPTER
808
USER INTERFACE ADAPTER
807
813
810

FIG. 8

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 1 265 161 A3

(12) EUROPEAN PATENT APPLICATION

(88) Date of publication A3:
28.04.2004 Bulletin 2004/18

(43) Date of publication A2:
11.12.2002 Bulletin 2002/50

(21) Application number: **02253371.5**

(22) Date of filing: **14.05.2002**

(51) Int Cl.[7]: **G06F 17/30**

(84) Designated Contracting States:
AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE TR
Designated Extension States:
AL LT LV MK RO SI

(30) Priority: **04.06.2001 US 874788**

(71) Applicant: **Hewlett-Packard Company**
Palo Alto, CA 94304 (US)

(72) Inventors:
- **Baskins, Douglas L.**
  Fort Collins, Colorado 80524 (US)
- **Silverstein, Alan**
  Fort Collins, Colorado 80525 (US)

(74) Representative: **Jehan, Robert et al**
Williams Powell
Morley House
26-30 Holborn Viaduct
London EC1A 2BP (GB)

(54) System for and method of storing data

(57) An adaptive digital tree data structure incorporates a rich pointer object (104, 105, 110-112, 114-118), the rich pointer including both conventional address redirection information (116B) used to traverse the structure and supplementary information (116A) used to optimize tree traversal, skip levels, detect errors, and store state information. The structure of the pointer is flexible so that, instead of storing pointer information, data may be stored in the structure of the pointer itself and thereby referenced without requiring further redirection.

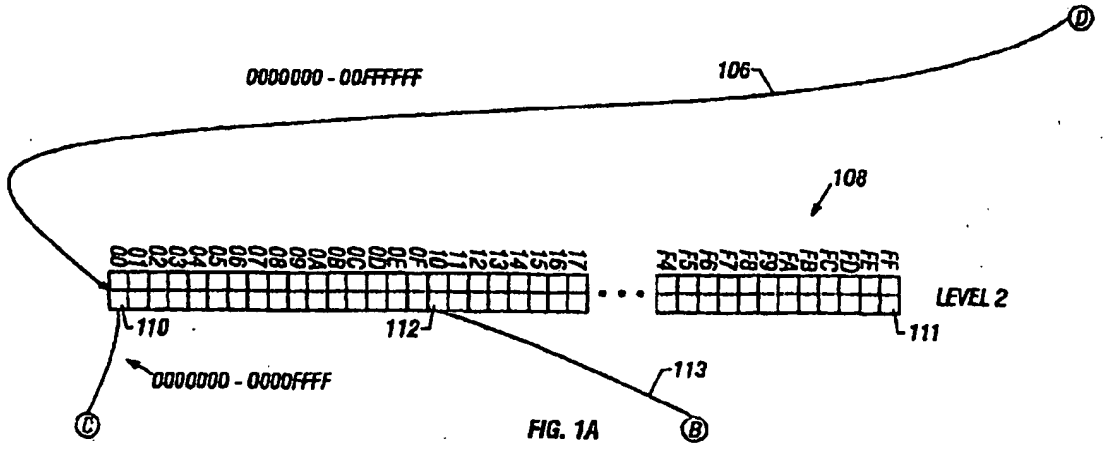


FIG. 1A

EP 1 265 161 A3

European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 02 25 3371

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.7)
X Y	US 5 848 416 A (TIKKANEN MATTI) 8 December 1998 (1998-12-08) * abstract * * column 4, line 5 - line 15 * * column 4, line 45 - line 63; figure 3 *	1,6,10 2-5,7-9	G06F17/30
Y	YAZDANI N ET AL: "FAST AND SCALABLE SCHEMES FOR THE IP ADDRESS LOOKUP PROBLEM", PROCEEDINGS OF THE IEEE CONFERENCE 2000 ON HIGH PERFORMANCE SWITCHING AND ROUTING. HEIDELBERG, GERMANY, JUNE, 26 - 29, 2000, PROCEEDINGS OF THE IEEE CONFERENCE ON HIGH PERFORMANCE SWITCHING AND ROUTING, NEW YORK, NY: IEEE, US, PAGE(S)' 83-92 XP001075689 ISBN: 0-7803-5884-8 * page 86, left-hand column, line 23 - right-hand column, line 18 * * page 90, left-hand column, line 51 - right-hand column, line 14 *	2-5,7-9	
A	WO 01 08045 A (SHADMON MOSHE; ORI SOFTWARE DEV LTD (IL)) 1 February 2001 (2001-02-01) * abstract * * page 14, line 14 - line 20 * * page 25, line 16 - line 21 *	1-10	TECHNICAL FIELDS SEARCHED (Int.Cl.7) G06F

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
BERLIN	27 February 2004	Moon, T

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)

ANNEX TO THE EUROPEAN SEARCH REPORT ON EUROPEAN PATENT APPLICATION NO. EP 02 25 3371

This annex lists the patent family members relating to the patent documents cited in the above-mentioned European search report. The members are as contained in the European Patent Office EDP file on The European Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

27-02-2004

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5848416 A	08-12-1998	AT 210856 T	15-12-2001
		AU 690282 B2	23-04-1998
		AU 2617495 A	04-01-1996
		CN 1152365 A	18-06-1997
		DE 69524601 D1	24-01-2002
		DE 69524601 T2	08-08-2002
		EP 0772836 A2	14-05-1997
		WO 9534155 A2	14-12-1995
		JP 2957703 B2	06-10-1999
		JP 10504407 T	28-04-1998
WO 0108045 A	01-02-2001	WO 0108045 A1	01-02-2001
		AU 4927599 A	13-02-2001
		CA 2380348 A1	01-02-2001
		EP 1208479 A1	29-05-2002
		JP 2003505791 T	12-02-2003

EPO FORM P0459

For more details about this annex : see Official Journal of the European Patent Office, No. 12/82

Modified LC-Trie Based Efficient Routing Lookup

Ravikumar V.C, Rabi Mahapatra and J.C. Liu
Department of Computer Science, Texas A & M University
{vcr,rabi,jcliu}@cs.tamu.edu

Abstract

IP address lookup at the router is a complex problem. This has been primarily due to the increasing table sizes, growth in traffic rate and high link capacities. In this work we have proposed an algorithm for fast routing lookup with reduced memory utilization and access time. This approach shows significant performance improvement in the average case and optimizes the overall time taken for packet routing. Since storage requirement, processing time and number of lookups performed are reduced, power consumption by the router is also reduced. Our simulation result indicates that the proposed technique works approximately 4.11 times better than the standard LC Trie approach in the average case.

1. Introduction

Due to the technology advancement, the packet transmission rate is ever growing. With increasing number of hosts and sessions in the Internet, the processing of packets at the routers has become a major concern. While the current technology supports fast switching, packet forwarding is still a bottleneck. This is either due to a high processing cost or large memory requirement [9].

The routing tables implemented in most routers today generally follow the software-based approach, as they are more flexible and adaptive to any changes in the protocol. The hardware solutions are also not scalable and hence with the emergence of the IPv6, these approaches are nearly impractical. The software approaches expect to gain speed as the processing rate is doubled every 18 months (Moore's law).

From early 90's to late 90's, the number of entries in the lookup table has changed from linear to super linear. In 2002, the backbone router contains approximately 100,000 prefixes and is constantly growing [9]. A lookup engine deployed in the year 2002 should be able to support approximately 400,000-512,000 prefixes in order to be useful for atleast another 3 years. Thus the lookup algorithms must be able to accommodate future growth.

Apart from the lookup entries the speed of the links are doubling every year. The links running on OC768c (approximately 40Gbps) require the router to process 125 million packets per second (Mpps) (assuming minimum sized 40 byte TCP/IP packets) [9]. For applications that do not require quality of service, a lookup or classification algorithm that performs well in the average case is desirable. This is so because the average lookup performance can be much higher than the worst-case performance. For such applications the algorithm needs to process packets at the rate of 14.1 Mpps for OC768clinks, assuming an average Internet packet size of approximately 354 bytes [9].

2. Research Objectives

The main motivation behind our approach is as follows:

1) Reducing the lookup time: Many of the trie-based approaches like [1] [2] take 6-7 memory references for lookup, while hash technique combined with Tries in [3] takes 5 memory references. Our goal is to modify the trie-based approach to reduce the number of memory references for each lookup.

2) Reduce routing table storage: Implementing the routing table using trie [4] could be an expensive process. In this work our aim is to reduce the storage requirements by eliminating unnecessary and redundant data especially when the number of nodes are large.

3) Ability to handle large routing tables: Most of algorithms support routing table sizes for the current needs. However an algorithm needs to be scalable so that it supports the routing table storage requirements for atleast the next 3 years.

Proceedings of the 10th IEEE Int'l Symp. on Modeling, Analysis, & Simulation of Computer & Telecommunications Systems (MASCOTS'02)
1526-7539/02 $17.00 © 2002 IEEE

3. Previous work on lookup algorithms

We have classified the existing approaches into Trie and Non-Trie based approaches. We briefly analyze the different approaches with respect to lookup time and memory utilization.

Table 1. Binary Strings to be stored in a Trie

Nbr	String	Nbr	String
1	000	11	0110
2	00101	12	0111
3	010	13	10100
4	011	14	10101
5	100	15	10110
6	101	16	10111
7	110	17	11101000
8	1110100	18	11101001
9	0000	19	101000
10	0001	20	101010

3.1. Non-Trie based approaches

Linear search is simplest data-structure with the linked list of all prefixes in the forwarding table. Storage and time complexity is O (N). With the emergence of IPv6 this approach is almost impractical to continue. Binary search algorithms include the classical methods like the hashing and tree based approaches. These techniques cannot distinguish matches based on prefix length. However [3] takes care of this constraint and proposes a modified binary search technique that uses $\log_2$ (2N), where N is the number of routing table entries. These algorithms are also computation intensive and require a large storage especially as the lookup table grows. Caching is a better technique than other memory reference techniques, as it is faster [10] [6]. A fully associative memory, or content-addressable memory (CAM) [8] [9], can be used to perform an exact match search operation in hardware in a single clock cycle. CAM implementation is a costly mechanism and is not feasible. Thus the storage requirements for these become a limiting factor. Also CAM based approaches are for fixed length prefixes. A better solution is to use a ternary-CAM (TCAM), a more flexible type of CAM that enables comparisons of the input key with variable length elements.

3.2. Trie based approaches

Trie [4] is a general-purpose data structure for storing strings. Each string (prefix) in the routing table is represented by a leaf node in the trie. The longest prefix search operation on a given destination address starts from the root node of the trie [9]. Patricia trie is a modification of a regular trie. The difference is that it has no one-degree nodes. Each branch is compressed to a single node in a Patricia tree. Thus, the traversal algorithm may not necessarily inspect all bits of the address consecutively, skipping bits that formed part of the label of some previous trie branch. Patricia tree loses information while compressing chains: the bit-string represented by the other branches of the uncompressed chain is lost. This is overcome by a path compression technique that records by maintaining a skip value at each node that indicates how many bits have been skipped in the path [5]. The statistical property of this trie (Patricia trie) indicates that it gives an asymptotic reduction of the average depth for a large class distribution [11]. However when the trie is densely distributed this approach fails in terms of storage and processing for lookup. Level compressed trie (LC-trie) [1] is a modification to this scheme for densely populated tries. An *LC-trie* is created by path compressing a binary trie and expanding every node rooted at a complete sub-trie of maximum depth to create a 2^k-degree node [9]. This expansion is done recursively on each subtrie of the trie. It replaces the i highest complete levels of the binary trie with a single node of degree 2^i. All the internal nodes represented in this trie contain no information but pointers to the first child. Hence a separate vector is needed to store all possible prefix in a *prefix vector* in case of a failure in search. Also since each node is traversed again by backtracking in case of failure this is an inefficient method both in terms of processing time and storage. Thus all the variants of the trie-based approach are not storage efficient and require a lot of processing.

4. Modified LC-Trie

In this section we describe a scalable time efficient level compression technique. The approach presented here is motivated to minimize the access time and memory utilization during routing table lookup, to transfer packets to the appropriate Ethernet port. A few approaches like [1] have been proposed to better storage and time complexity. Our approach is based on the LC-trie approach for compressing the trie. We use the same compression technique as in LC-trie. However we try to avoid additional storage (LC-Trie uses additional data structures like base, prefix and nexthop vector to store prefixes) and processing (like backtracking), which are the major flaws of LC-Trie approach. Figure 1 represents the tree corresponding to Table 1 for the proposed approach. From the Figure 1 we see

that unlike the LC-trie approach our approach stores all the prefixes either in the internal nodes or leaf nodes.

The algorithm first converts the routing table entries into a binary trie. Then path compression is done on this trie to reduce its depth. This path-compressed trie is now level compressed by storing the sub strings at the internal nodes and the strings at the leaves. When the routing table is built we use a FILL FACTOR (this represents the maximum number of branches that each node can have during the build) to help make the future updates easier.

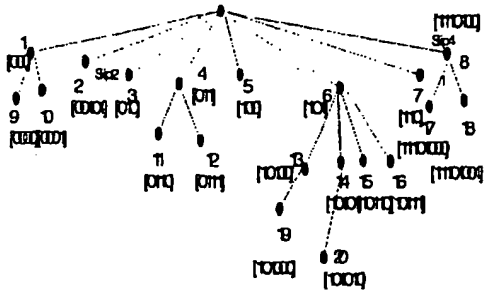


Figure 1. Scalable Time Efficient Level Compression

4.1 Storage data structure

Each node of our trie is represented as in Figure 2. Following is a brief description of the significance of each of the fields in the data structure.

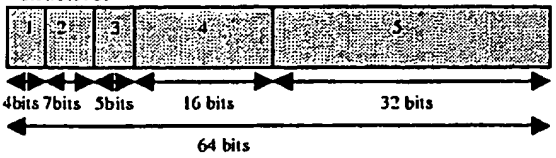


Figure 2. Proposed Data structure for node in routing table (IPv4)

Branching factor [0:3]: This indicates the number of descendents of a node. This is a 4-bit value and a maximum of 16 branches to a single node can be represented.

Skip value [4:10]: This indicates the number of bits that can be skipped in an operation. This is a 7-bit value and a maximum of 128-bit skip can be represented.

Port [11:15]: This represents the output port for the current node in case of a match. This is a 5-bit value and this field can represent a maximum of 32 output ports.

Pointer [16:31]: This is a pointer to the leftmost child in case of an internal node and NULL in case of a leaf node. This is a 16 bit value and can represent a maximum of 65536 prefixes. The current implementation assumes the number of routing table entries to be less than 65536. However a scalable solution to consider more than 5,000,000 prefixes is discussed later.

String [32:63]: This represents the actual value of the prefix the node represents. The current implementation assumes a 32-bit value (IPv4) though it can be extended to 128-bit value (IPv6).

4.2 Algorithm

```
node = T[0]; s = ieeedata[k]; node = table->trie[0];
pos = GETSKIP(node); branch = GETBRANCH(node);
adr = GETADR(node);
prefix=0; result= -1; /* stored in Register */
while(branch != 0) /* Not leaf node */
{   node = table->trie[adr + EXTRACT(pos,branch,s)];
    if(pos)
        prefix<<pos; /* skip 'pos' bits of the prefix */
    if (branch > n)
        prefix= prefix << (m+1)| branch;
        /* n=2**i-1 m is the number of bits
        representing the branch */
    if(GETSTRING(node)^prefix)
        break; /* Previous node contains largest prefix and
        interface stored in result */
    else
    {   pos = pos + branch + GETSKIP(node);
        branch = GETBRANCH(node);
        adr = GETADR(node);
        result = GETPORT(node);}
}
```

The search algorithm forms the bottleneck of the entire routing process and hence this needs to be designed very efficiently. Algorithm discussed above is used to search a string *s* in the routing table. EXTRACT (p,b,s) is used to search *s* in the routing table, where *b* is the number of bits starting at position *p*. Let the array representing the tree be T. The root is stored in T[0].

Each entry in Table 2 represents a node in the proposed approach, for routing table described in Table 1, with the corresponding branch, skip and pointer values. In addition to these three fields each node also has a 32-bit (IPv4) prefix represented by the node, which is not indicated in the table.

4.3 Working of the algorithm

The working of this algorithm is illustrated with an example. Consider the input string 101001. We start from root node number 0. We see that the branching factor is 3 and skip value is 0 and hence extract 1st 3 bits from the search string. These 3 bits represent a value of 5, which is added to the pointer, leading to position 6 in the

Proceedings of the 10th IEEE Int'l Symp. on Modeling, Analysis, & Simulation of Computer & Telecommunications Systems (MASCOTS'02)
1526-7539/02 $17.00 © 2002 IEEE

array. At this node the branching value is 2 and the skip value is 0 and hence we extract the next 2 bits. They have the value 0. However we check if the string (101) matches the prefix (101). Since it matches the search continues further. We now add the value of 0 to the pointer and arrive at position 13. At this node the branching factor is 1 and skip value is 0. They have a value of 1. We again compute to see if the string (10100) is same the prefix (10100). Since it matches we continue and add the value of 1 to 19 to obtain the pointer 20. Now see that this node represents a leaf node since the branching factor is 0. We now check to see if the string (101001) matches the prefix (101011). Since they don't match we use the previous value of the output port from the register to route the packet. The Figure 3 represents the path taken during the search.

Table 2. Array representation our approach

	branch	skip	pointer		branch	skip	pointer
0	3	0	1	11	0	0	0
1	1	0	9	12	0	0	0
2	0	2	0	13	1	0	19
3	0	0	0	14	1	0	20
4	1	0	11	15	0	0	0
5	0	0	0	16	0	0	0
6	2	0	13	17	0	0	0
7	0	0	12	18	0	0	0
8	1	4	17	19	0	0	0
9	0	0	0	20	0	0	0
10	0	0	0				

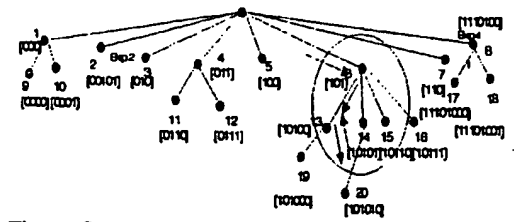


Figure 3. Trie traversal for the string 101001

Thus we see that this approach doesn't traverse the entire trie in case the string is not present. Also there is no separate storage for the prefixes as in case of LC-Trie. This approach checks for the match in the string at every step. However this is not computation intensive since the string to compare is already present in the cache and a *xor* on the bits could give us the result of comparison.

In the LC-trie approach after we traverse through the trie we perform a check for the string match in the base vector, which uses hashing technique, consuming at least one memory fetch. If there is a mismatch a check is done again on the prefix table and this requires hashing to check for a prefix match, which again requires another memory fetch. Thus compared to the LC-trie we save atleast two memory cycles for every routing lookup performed.

4.4. IPv6 Compatibility

The algorithm can easily be extended to IPv6 and to allow a maximum of 2^{37} entries. This ensures that the proposed data structure can support routing entries beyond 2005. This data structure also allows handling of a maximum of 1024 interfaces. The data structure for each node is described in the Figure 4.

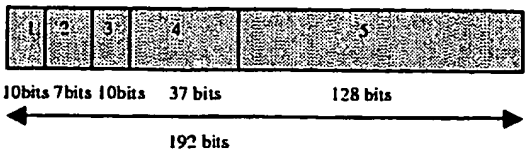


Figure 4. Proposed Data structure for node in routing table (IPv6)

5. Simulations & Experimental Setup

To test and verify our approach with the LC-Trie approach we have modified the test bed used by the authors of [1]. The features of this modified test bed are as follows:

1. It reads routing data from the routing table file, which is in a predefined format as discussed in the paper [1]. The routing file is an exhaustive list of routing entries (65536 entries for 16-bit pointer value).
2. The algorithm can be run over a number of times by specifying *n* as a command line argument. As the number of iterations increase, it gives a good estimate of the parameters under comparison.
3. Quick sort algorithm is used to sort the routing table entries.
4. We have also used two different approaches to compare the performance. One uses a function call to the search algorithm and the other is an inline function. However we have used the inline function results for our comparison.

In our implementation we have used routing tables similar to that provided by the Internet Performance Measurement and Analysis project

obtained is an estimate of power for one iteration. The results show that proposed approach and the LC-Trie approach consumes 4.615mW and 4.755mW for 20 lookups respectively (Table 5). This is a reduction of 0.14mW for 20 lookups. This is directly related to the fact that the time for lookup is less. The routing entries in our simulations are assumed to be stored in the DRAM and the storage power corresponding to that was computed for both the approaches. Since the storage requirements are reduced by factor of 2.38, it is expected that power consumptions will be less by that amount.

Table 3. Timing

Parameters	LC-Trie	Proposed	Savings(%)
B_w	10.57 sec	0.43 sec	24
N_w	0.05 sec	0 sec	100
S_w	0.37 sec	0.36 sec	2.7
cP_{acc}	5.01 sec	1.53 sec	69.4
P_{acc}	5.02 sec	1.53625 sec	69.4
P_{mr}	0.01	0.0074402	
P_{ac}	1308104	4283399	
I_{acc}	4.12 sec	1.0 sec	75.7
I_{avg}	4.12 sec	1.005 sec	75.7
I_{mr}	0.01	0.0053452	
I_{ac}	1590680	6553600	

Table 4. Memory Utilization

Parameters	LC-Trie	Proposed
B_w	32769*16	0
P_w	32767*14	0
N_w	4*16	0
T_w	655374	655378

Table 5. Power consumed

Approach	No cycles	Proc. Config	Power/20lookup
LC-Trie	5944854	0.8mW/Mhz	4.755 mW
Proposed	5769630	0.8mW/Mhz	4.615 mW

7. Conclusion & Future work

We have proposed a scalable and efficient algorithm for fast lookup based on modified LC-Trie technique. This algorithm performs about four times better in terms of access time in the average case as compared to the LC-Trie approach.

The proposed algorithm does approximately 6.6 million lookups per second on a 32 bit, 200Mhtz processor without considering caching of packets. Since the packet have certain amount of locality in them, caching could lead to better performance. The search algorithm forms the bottleneck in the lookup process. Hence computation kernels can be obtained and optimized by implementing at the hardware level. One such method is by creating tie instructions (in Xtensa) for the set of instructions that are executed more frequently.

8. References

[1] S. Nilsson and G. Karlsson, Fast Address Look-up for Internet Routers, Proceedings of IEEE Broadband Communications 98, (April 1998), 9-18.

[2] Venkatachary Srinivasan and George Varghese, Faster IP Lookups Using Controlled Prefix Expansion, Measurement and Modeling of Computer Systems, (1998), 1-10.

[3] Marcel Waldvogel, George Varghese, and Jon Turner and Bernhard Plattner, Scalable High Speed IP Routing Lookups Proceedings of SIGCOMM '97, (1997), 25-36.

[4] Edward Fredkin. Trie memory. Communications of the ACM, (1960), 490-500.

[5] W Szpankowski, Patricia Tries again revisited, Journal Of the ACM '90, (Oct 1990 Vol.37, No. 4), 691-711.

[6] Tzi-Cker Chiueh and Prashant Pradhan, "Cache Memory Design for Network Processors", HPCA, (2000), 409-418.
http://ipdps.eece.unm.edu/2000/raw/18000884.pdf/

[7] "Internet Performance Measurement Analysis Project", University of Michigan and Merit Network, [Online]: http://www.merit.

[8] Tzi-Cker Chiueh and Prashant Pradhan, High-Performance IP Routing Table Lookup Using CPU Caching, INFOCOM '99, (1999), 1421-1428.

[9] Pankaj Gupta "Algorithms for routing lookups and packet classification", A dissertation submitted to the department of computer science and the committee on graduate studies of Stanford university, Dec 2000,[Online]:
http://klamath.stanford.edu/~pankaj/thesis/thesis_2sided.pdf

[10] Tzi-cker Chiueh and Prashant Pradhan, Suez: A Cluster-Based Scalable Real-Time Packet Router, International Conference on Distributed Computing Systems, (2000), 136-143.

[11] P.Newman, G.Minshall, T.Lyon, and L.Huston, IP switching and gigabit routers. IEEE Communications Magazine, 35(1): (January 1997), 64-69.

Proceedings of the 10th IEEE Int'l Symp. on Modeling, Analysis, & Simulation of Computer & Telecommunications Systems (MASCOTS'02)
1526-7539/02 $17.00 © 2002 IEEE

[7]. In order to compare the modified technique with LC-Trie approach the traffic was simulated and we used a random permutation of all possible entries in the routing table. The time measurements have been performed on sequences of lookup operations, where each lookup includes fetching the address from the array, performing the routing table lookup, accessing the nexthop table and assigning the result to a volatile variable. Some of the entries in the routing tables contain multiple nexthops. In such cases we select the first one listed as the nexthop address for the routing table, since we only consider one nexthop address per entry in the routing table. However for entries that didn't contain a nexthop address a special address that is different from the ones found in routing table was used.

The following equations were used in the computation of average and standard deviation of the samples (t_i).

Average Time (avg) = t_i/n

Std Deviation (std) = $(t_i^2 - n*avg*avg)^{1/2}/(n-1)$

Parameters analyzed:

We have analyzed the effectiveness of our approach and compare our approach with the LC-Trie approach with respect to timing, storage and power consumption. The parameters considered in each of the cases are described below.

5.1. Timing

5.1.1 Building. Time taken to Build Routing table (B_t): This is the time taken for the algorithm to retrieve all the data from the Routing table file, sort them and build them with appropriate entries for future referencing.

Time taken to build nexthop table (N_t): This is the time taken to compute all the next hop addresses from the routing table data.

5.1.2 Sorting. Time taken to Sort the entries (S_t): Based on a seed value the routing table entries are stored in a temporary data structure in a random fashion, which is then sorted for building the routing table.

5.1.3 Searching.

Function Search: Time taken to search the string (using call to a function) based on *n* iterations.

F_{min}: Minimum time taken to search the string using function call.

F_{avg}: Average time taken to search the string using function call.

F_{std}: Standard deviation of the times for searching a string using function call.

F_{lps}: Average number of lookups/second using function call.

Inline Search: Time taken to search the string (using an inline function) based on *n* iterations.

I_{min}: Minimum time taken to search the string using Inline function call.

I_{avg}: Average time taken to search the string using Inline function call.

I_{std} : Standard deviation of the times for searching a string using Inline function call.

I_{lps}: Average number of lookups/second using inline function call.

5.2 Memory Utilization

B_m: Memory utilization in bytes for the base vector.

P_m: Memory utilization in bytes for prefix vector.

N_m: Memory utilization in bytes for nexthop vector.

Trie (T_m): Memory utilization for Trie.

6. Results

Following are the results for the comparison of LC-Trie approach and proposed approach with a fill factor of 0.5 (this is a good value based on the experimental results considering future updates) and a fixed branch at root (16). We have run this algorithm 100 times to get a good estimate of the values. This was run on an Intel Pentium II processor, 400Mhz and 256 MB RAM. The programs were written in C and complied with gcc compiler using optimization level –04.

From Table 3 we observe that time taken to build the trie is reduced by 0.14 seconds. This is mainly due to the fact that no additional computation is required to build the base and prefix vector. Also there is no overhead of building the nexthop table. The simulation results show that the proposed approach works 3.28 times better than LC-Trie approach when the prefix search is implemented as a function search and 4.11 times better when implemented as an inline function. Thus, for above mentioned system configuration we were able to achieve a lookup of approximately 6.6 Mpps in the average case. From Table 4 we observe that the proposed approach avoids the storage for base, prefix and nexthop vector and hence occupies 2.38 times lesser storage. Though the reduction in storage for the nexthop vector is not significantly high, the storage for the base and prefix vector is greatly reduced.

The processing power savings for the two approaches were compared using an implementation based on reconfigurable processor architecture from *Tensilica* (16/24 bit Xtensa ISA, 200Mhz, 0.18 um technology, 0.7 mm^2 core area, 0.8mW/MHz core power dissipation). The Routing table used in our power analysis is described in Table 1. The result